A product line architecture for web-based data management systems

Janne Ruuttunen

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyopin laboratorio

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory of Information Processing Science

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF MASTER'S THESIS

Author:	Janne Ruuttunen
Name of the thesis:	A product line architecture for web-based data management systems
Date:	August 30 2004 **Pages:** ix + 57
Dept.:	Department of Computer Science and Engineering
Professorship:	T-106 Software Systems
Supervisor:	Prof. Jorma Tarhio
Instructor:	Jukka Aunu

Reuse of software in the form of software product lines is one of the most promising ways to improve the efficiency of software engineering. The product-line approach has been found to be suitable not only for companies producing commercial software but also for pure consultancies where the business is based on project contracts instead of producing software for public market.

In this work, the suitability of the product-line approach is examined in a small consultancy that has a lot of experience in producing diary systems for governmental organizations. The work is based on a framework prototype, designed for participating to a bid contest, that later on evolved into a full-blown software product line.

The thesis consists of a cross-section to the central techniques of the product line and the theoretical background thereof. In particular, the product-line architecture, object-oriented frameworks and their extension models, and object-relational mapping through a persistence layer are examined. Finally, the project is evaluated from organizational and technical points of view.

Keywords: software product line, product-line architecture, object-oriented framework, object-relational mapping, persistence layer

TEKNILLINEN KORKEAKOULU

DIPLOMITYÖN TIIVISTELMÄ
(Finnish abstract)

Tekijä:	Janne Ruuttunen
Työn nimi:	A product line architecture for web-based data management systems
Päivämäärä:	30.8.2004 **Sivumäärä:** ix + 57
Osasto:	Tietotekniikan osasto
Professuuri:	T-106 Ohjelmistojärjestelmät
Valvoja:	Prof. Jorma Tarhio
Ohjaaja:	Jukka Aunu

Ohjelmistojen uudelleenkäyttö tuotelinjojen muodossa on lupaavimpia tapoja tehostaa ohjelmistotyön tuottavuutta. Tuotelinjalähestymistavan on todettu soveltuvan paitsi ohjelmistotuotteita valmistaville organisaatioille myös konsulttiyrityksille, joissa liiketoiminta perustuu ainoastaan asiakasprojekteihin, ei omiin ohjelmistotuotteisiin.

Työssä tarkastellaan ohjelmistotuotelinjalähestymistavan soveltuvuutta pienessä konsulttiyrityksessä, jolla on vankka kokemus valtionhallinnollisten diaari- ja asianhallintajärjestelmien tuottamisesta. Työn pohjana on tarjouskilpailua varten kehitetty ohjelmistokehysprototyyppi, joka sittemmin kehittyi täysipainoiseksi ohjelmistotuotelinjaksi.

Työssä tehdään läpileikkaus ohjelmistotuotelinjan keskeisiin tekniikoihin ja niiden teoreettisiin perusteisiin, sekä sen kehitysprosessiin. Huomiota kiinnitetään erityisesti tuotelinja-arkkitehtuuriin, olio-ohjelmistokehyksiin ja niiden laajennusmalleihin sekä olio- ja relaatiomallien välisen kuvauksen toteuttamiseen ns. persistenssikehyksen avulla. Lopuksi projektia arvioidaan organisatorisista ja teknisistä näkökulmista.

Avainsanat: ohjelmistotuotelinja, tuotelinja-arkkitehtuuri, oliopohjainen ohjelmistokehys, kuvaus oliomallista relaatiomalliin, persistenssikerros

Acknowledgements

I want to take this opportunity to thank my employer Oikeat Oliot Oy and my foreman and supervisor Jukka Aunu for professional and financial support in the preparation of this thesis. I also want to thank my colleague Antti Pirilä for cooperation and inspiring conversations about the design and philosophy of the Alfred product line. Laura Juntunen's style tips and comments have also been of great help.

Helsinki, August 30 2004

Janne Ruuttunen

Contents

List of Figures

Terms and Abbreviations

API Application Programming Interface

COTS Commercial Off-The-Shelf

CRUD Create, Retrieve, Update, Delete

Hot spot A specific extension point in a framework

DDL Data Definition Language

HTTP HyperText Transfer Protocol

IDE Integrated Development Environment

JDBC Java Database Connectivity

PLA Product-line Architecture

RDBMS Relational Database Management System

SA Software Architecture

SPL Software Product Line

SQL Structured Query Language

(G)UI (Graphical) User Interface

WYSIWYG What You See Is What You Get

XML Extensible Markup Language

Chapter 1

Introduction

1.1 Background

In the Finnish governmental administration, the software acquisition procedure rests upon the traditional assumption that the client organization can specify a system in advance, get bids for its construction and have it built. Given the inherently complex nature of software systems [9], this rather controversial practice makes it difficult to establish a fair competition between the contractors, and is not likely to yield the economically and qualitatively best possible solution for the client.

Beyond the system acquisition context, incremental software processes have long been known to be more efficient than the traditional waterfall process model [25], but an impartial, incremental procedure for governmental software acquisition would be hard to imagine. This being the situation, software contractors can only try to improve their internal software processes and bid as low as they can – or have friends on the client side.

Software reuse is one way of decreasing the costs of software development. This has been clear as long as computer software has been developed; hundreds of articles have been written about reuse, but with surprisingly poor results [22]. Indeed, it seems that there are many software organizations where the benefits of software reuse are not being successfully exploited, and many where the possibilities to a more effective reuse of existing assets are being researched.

This thesis deals with one such attempt and is based on my work at Oikeat Oliot Oy, a small Helsinki-based company providing computer consultancy services. Oikeat Oliot Oy has been involved with diary system design, development and maintenance for over ten years, and therefore has a lot of experience about the challenges of that domain. The software system to be presented was originally designed and developed for participating to a bid contest set out by a Finnish governmental organization in order to produce a new diary system.

1.2 Motivation

Constructing software systems by composing components has been regarded as the most promising approach to increase the productivity of software engineering efforts [22]. A software product line brings this idea further and defines an explicitly defined software architecture, i.e. a product-line architecture, that defines how the components are composed and configured. Individual products are then derived from the product line according to the configuration model of the product line. The level of reuse rises and its scope broadens, because not only program code is reused but for instance the build process, unit tests, documentation etc.

The idea is intuitively attractive for nearly all kinds of software organizations, and indeed, in the past ten years, many organizations have reported significant savings in software development costs because of the product-line approach (for a survey, see for example [12]).

Software product lines are usually thought to fit primarily to the business model of software companies or embedded systems (e.g. consumer electronics) manufacturers [8]. In this work, however, the concepts and ideas are applied to a smaller-scale, consultancy type business where the focus is on creating tailored software systems by contract, not producing software products, product families or embedded systems for public market.

In the business of producing software systems for a particular domain, e.g. diary systems for governmental organizations, there are many interesting commonalities, both technically and functionally, that irresistibly call for the product-line approach.

1.3 Objectives

What are the advantages and disadvantages of using the product-line approach in small software consultancies? What constitutes a software product line in practice? How can a software product line be organized? Wouldn't an object-oriented framework suffice? What is the essence of a software component?

This work aims to answer these questions through a concrete product line project called Alfred. The presentation is a cross-section to the design of Alfred, beginning from the product-line organization and component model to the implementation of a persistence framework component. Also, two Alfred-based applications are presented.

In summary, the following list recapitulates the objectives of this thesis:

1. Document the development process and history of the Alfred Product Line
2. Describe the current status and assess the design decisions made
3. Create insight and valuable experience for future adoption of the product-line approach and the various technical aspects thereof

This thesis also attempts to clarify the rather confusing terminology used in conjuction with software architectures, software product lines and object-oriented frameworks through concrete, non-trivial examples. Such examples are few in the scientific literature encountered during the preparation of this thesis.

1.4 Organization

This thesis is organized as follows. Chapter 2 starts with an introduction to software architecture. Next, software product lines and their architectures are examined, followed by a introduction to object-oriented frameworks, a common building block of software product lines. An interesting domain for object-oriented frameworks, namely object–relational mapping and persistence layers, will be discussed in order to provide concepts for an essential part of the Alfred Product Line presentation given in chapter 3.

In chapter 4, two Alfred-based application prototypes are discussed: a diary system framework and an application form delivery system. Finally, conclusions are drawn in chapter 5, along with a quality evaluation based on past experiences, followed by some future directions.

This thesis assumes that the reader is already familiar with the object-oriented paradigm and its concepts and has a basic knowledge of the UML notation.

Chapter 2

Background

The concepts used in this thesis are introduced in this chapter. The first section is an overview of Software Architectures. An interesting application of software architectures is the so-called product-line architecture. A product-line architecture defines a Software Product Line, which is the subject of the second section. In the third section, the focus narrows to object-oriented frameworks, a common building block of software product lines. Finally, the fourth section discusses the common software domain of object-relational mapping and its implementation as an object-oriented framework.

2.1 Software Architectures

The bigger and more complex our software systems grow, the bigger is our need for informative descriptions of the software. A discipline called Software Architecture (SA) addresses this issue. But what do we mean by software architecture? The architecture of a software system is concerned with the top-level decomposition of the system into its main components. One definition of software architecture is given in Bass *et al.* [5]:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them.

An explicitly specified architecture of a software system is important for three reasons. First, the choice of a certain software architecture theoretically imposes upper or lower limits to the quality attributes of a system, such as performance, reliability, or maintainability. Second, it allows for communication about the software product early on in the development process. The third reason, when applicable, is that software architecture defines the shared components of a software product line. [8]

2.1.1 Concepts

A central concept of software architecture is that of a *component*. Jan Bosch defines a software component as "a unit of composition with explicitly specified provided, required and configuration interfaces and quality attributes" [8]. This definition differs somewhat from more traditional definitions in that a component is not thought to be capable of composition by third parties without adaptation, which is an appropriate restriction for our model as well (see section 3.2.1).

The term *component* is also ambiquous with respect to its fundamental nature. Some SA authors understand components as run-time entities, resembling components in the physical world, e.g. parts of machinery. Furthermore, in distributed systems literature, the term *component model* is sometimes used to refer to the various distributed object models such as OMG's Common Object Request Broker Architecture [14] (CORBA) or Enterprise JavaBeans [19]. In the remainder of this thesis, the term *component* shall bear the meaning defined above, as a static unit of software composition.

Generally, a software component implements a *domain*. A domain is a logical and recognized field of functionality. While a software system or application is typically associated with an *application domain*, e.g. banking or building automation, software components frequently cover a *software domain*. Typical examples of software domains include user interfaces, lexing and parsing, and communication protocols. [8]

Bosch also notes that a domain is frequently represented by one primary component and several smaller, secondary components that handle the variation in functionality required from the component. In chapter 3.2.1, this idea is associated to the notion of *abstract* and *extension components*.

Domains themselves can sometimes be decomposed into lower-level domains, and their implementation delegated to corresponding components, according to the software architecture. For example, a technical communications domain can be decomposed into layers of more specialized technical domains. An example of this is discussed in section 2.1.2 in conjunction with the layered architectural style.

Despite the fact that there are plenty of generally recognized domains, there are few standards on taxonomies for domains, be it application or software domains. The most notable of these are the works of OMG [39], including standards for business objects, electronic payments, and medical facilities.

Components generally interact through *connectors*. Common types of connectors include procedure calls, event broadcast, database protocols and pipes. [49]

2.1.2 Architectural Styles

Architectural styles are patterns of structural organizations of components and connectors. There are plenty of commonly recognized architectural styles, such as Pipes and Filters, Layers, Blackboard (sometimes called 'Repository'), Interpreter, and

Process Control. Even object-orientation and implicit invocation are perceived as architectural styles by many authors. [49]

Below, the most relevant one to this thesis is discussed.

Layers

The essential ideas of the Layered style are the following [32]:

- The large-scale logical structure of a system is organized into discrete layers of distinct responsibilities with a clean separation of concerns such that the lower layers are low-level and general services, and the higher layers are more application specific. The layers are the main logical components of the system.
- In each layer, there should be dependencies only to lower layers, not higher. Depending on the nature of the system, it may or may not be appropriate to restrict the communication to adjacent layers only. In information systems, the standard is a "relaxed layered" architecture. The ways of communication between the layers represent the connectors in the Layered style.

An obvious example of a layered architecture is the OSI seven-layer model for communication protocols [53].

A prime disadvantage of the layered style is related to performance. Because the layered style organizes computational tasks based on level of abstraction, an external event typically causes the functionality to be divided over multiple layers. This in turn may cause performance problems. [8]

Two other widely known examples of the layered style are the classic two-tier and three-tier architectures.

The tiers in two-tier architecture refer to the user interface layer that directly accesses the data storage layer. In this arrangement all application logic is embedded in the user interface, e.g. in the window definitions, or possibly in database procedures. The obvious disadvantage of the two-tier system is the high coupling of very different concerns: user interface layout code and database access code. This leads almost certainly to an inflexible, unmaintainable system. [32]

A typical description of the vertical layers in a three-tier architecture is:

1. Interface — e.g. windows, reports
2. Application Logic — task and rules that govern the process
3. Storage — persistent storage mechanism

The basic idea is that the interface and storage layers should be free of application logic, and respectively, there should be no user interface or storage related code in the middle layer, sometimes also called the business layer. [32]

However, in this thesis the term *layer* will only refer to the abstraction layers, as opposed to the *tiers* as in an n-tier architecture[1].

2.1.3 Use of Software Architecture

Bosch [8] notes that software architectures are generally defined for three different purposes: as individual systems, as software product line architectures, or as standard architectures used for public component market. Where a software domain is commonly identified and tends to be implemented as a component over and over in software systems, it becomes subject to standardization work, and possibly ends up as the third kind, a standard architecture. While this is, according to Bosch, currently more an ambition rather than the state of practice, there are well known examples of this, including authentication mechanisms, object-relational mapping (a few standards of whom are introduced in section 2.4.3), and distributed objects technology. In this work, however, the focus is on the second purpose, namely on product-line architectures.

2.2 Software Product Lines

In the previous section, software product lines were identified as a central purpose for software architecture definition through the concept of product-line architectures. A software product line is defined by Paul Clements and Linda Northrop from the SEI (Software Engineering Institute in Carnegie Mellon University) [12] as follows:

> A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

Software product lines attack the costs of software development through reuse, which has traditionally been concerned with relatively small pieces of program code, e.g. algorithms or classes. In the product-line approach, however, an organization can reuse not just software, but also development environments, requirement analyses, test cases, and ideally any processes that relate to some phase of the product's life cycle. These reusable assets that constitute the basis for the software product line are called *core assets*. [12]

Each product in an SPL is formed by taking applicable components from the base of core assets, tailoring them as necessary through preplanned variation mechanisms such as parameterization or inheritance, adding any new components that may be necessary, and assembling the collection according to the rules of a common product-line architecture. These rules constitute the *configuration model* of the product line.

[1] Except for the term "persistence layer"; see section 2.4.3.

The applicability of software product line concepts might seem to be limited to sofware organizations that develop and market systems or products. But as Bosch [8] notes, the approach is also applicable for software consultant organizations developing software on project basis for other organizations, even though these kinds of organizations generally are more project than product oriented. This is due to the observation that many such organizations tend to perform projects in a particular domain, where the commonality between the projects allows for the development of a software architecture and a set of components that can be used for subsequent projects.

2.2.1 Benefits

From the broader perspective of an organization's productivity, software product lines offer a number of benefits. According to Clements and Northrop [12], these include the following:

- Product requirements often consist of a common requirements base, by extending it with some product-specific requirements. Thus a comprehensive requirements analysis is saved for each product.
- An architecture for a software system represents a significant investment of both time and talent from the organization. As a product-line architecture, it will be used for each product, and considerable time and risk are spared.
- Performance models and associated analyses are existing product line assets. With each new product, it is likely that that any problems related to e.g. concurrency have already been solved. The more complex the system, the higher is the payoff for solving these aspects once for the entire product line.
- Generic test plans, test processes, test cases, test data, and the communication paths required to report and fix problems are already in place. They only need to be tailored and possibly extended for the individual applications.

Previous experiences are of high value when estimating the effort of a software project. With product lines, the production plans and realizations of the previous projects offer a reliable basis for planning and effort estimation. Similarly, configuration management tools and procedures, and the overall development process are in place. They have been used before, and are reliable and responsive to the organizations needs. Moreover, fewer people are required to build products, and the people are more easily transferred to other projects across the software product line.

A switch to a product-line approach is not without consequences for *individual staff members* either. From the management perspective, the organizational benefits discussed above apply at the individual level also, because of the very purpose of management in organizations. But for the individual software developers and architects, there are numerous advantages as well.

In interviews conducted by Clements and Northrop [12], software product line developers pointed out advantages including the following:

- It is more interesting to focus on the truly unique aspects of the products rather than re-inventing the wheel over and over again.
- There are no difficult software integration phases because an already validated architecture is used, and the integration of the components has already been tested.
- There are fewer stressful schedule delays interfering with the developers' private lives because a proven production plan is followed.
- The developers are more marketable because of their knowledge of product line practices.
- The developers have greater mobility within the organization because their knowledge applies to all of the product line members.
- More of the developers' time is spared e.g. for getting involved in new, interesting technologies.

The development of the standard components is more challenging, but also more rewarding, because the work will effect many systems and have more users than an individual product component. It is also easier to sell the products if there is a reference to another product built from the product line (and even more so if the reference is in the same user organization), because most of the relevant metrics and features of the product are known or deducible in advance.

From the *customer's* point of view, the product-line approach is desirable because the product will have better quality and fewer defects causing delays and budget overruns. Well-tested training material and documentation will be available. The maintenance costs of the systen can be shared with other customers of the product line. In some cases, the customer may even be able to influence the evolution of the product line and its features.

Finally, what about *end users* who are not the paying customers of the system? To have fewer defects in the system is in their interest also, as are better end-user documentation and training materials. Furthermore, a product line typically reuses the user interface paradigms, so the end-user training needs only be arranged for a single member of the product family in the user organizations where a product line is used in implementing multiple systems.

2.2.2 Costs

Not surprisingly, there are costs related to the benefits arising from the product-line approach. The use of the commonalities in the core assets makes the creation and

maintenance of that asset much harder. For example, the software components must be made more robust than would be necessary in a one-off product [8]. In addition, the generality of the component should not imply loss of performance. The software architecture of the product line must support the variation inherent in the product line, which imposes an additional constraint on the architecture and requires greater talent to design it [12].

For software consultant organizations utilizing the product-line approach, the copyright issues need to be carefully addressed. In standard contracts, the client organization usually obtains the copyright on the developed system, whereas the developing organization gives it up. In order to employ the product-line approach, it is necessary that the developing organization retain the right to use and further develop the software, even though the ownership can be non-exclusive. To achieve this situation, the organization may need to offer some extra benefits to the customer, e.g. lower price for the project. [8]

2.3 Object-oriented Frameworks

While in many ways similar as concepts, object-oriented frameworks are generally not software product lines. The concept is much more limited and only concerns the software implementation of some, usually technical, domain. We now turn to object-oriented frameworks and discuss their relation to software product lines.

What is a framework? Perhaps the most widely known definition of a framework is the one by Johnson and Foote [30]:

> A framework is a set of classes that embodies an abstract design for solutions to a family of related problems.

In other words, a framework provides an implementation for the core and unvarying functions, and includes a mechanism to allow a developer to plug in the varying functions, or extend the functions. A framework can be designed to cover a certain part of a system, such as the user interface or database connectivity, or to provide the abstract design for the entire application [24].

Object-oriented frameworks has been a subject of enthusiastic research since the uprise of object-oriented languages in the early 80's, and numerous articles have been published about software reuse in form of object-oriented frameworks. As is the case with any subject of experience reports, the failures get less attention for obvious reasons. For positive experiences see for example [13, 11].

By observing the evolution of computer languages and available software systems, it can be seen that as the abstraction level of the implementation tools rises, along grows the size and functionality of systems that can be implemented and maintained. The same applies to framework design: the effort of generalizing the phenomena and

functions of the domain has the very purpose of rising the abstraction level of the tools provided for the application developer.

2.3.1 Concepts

Classification of frameworks

Frameworks can be roughly divided in two categories: *black-box* frameworks and *white-box* frameworks. These terms are familiar from other fields of software engineering and refer to the distinction of whether the implementation details are available to, in this case, the application developer. However, in the context of frameworks, this interpretation is not quite accurate, as explained below.

White-box frameworks are based on inheritance. Application development using a white-box framework consists of sub-classing framework classes and overriding their operations. The relationship of framework objects and application-specific objects is defined statically at compile-time. With black-box frameworks, applications are developed by parameterizing and composing framework objects. Object compositions can be changed dynamically. [20]

It should be noted that in white-box frameworks, the implementation of the framework classes does not necessarily need to be available, just the interface specification does. Similarly, in a purely composition-based (black-box) framework, it may be necessary to have the implementation of a framework class available, for example, for correct parameterization. In practice, frameworks employ a mixture of the above techniques, and as noticed by Johnson and Foote [30], white-box frameworks tend to evolve into black-box frameworks.

Another basis for framework categorization is the direction of the communication. *Called* frameworks are used as class libraries, i.e. by calling the framework objects' operations as needed, whereas in *calling* frameworks, the framework classes control the execution and call their abstract operations, whose implementation is provided by the actual application code. [51]

Object-oriented frameworks are not Software Product Lines on theirselves, but they can be used to construct them, as mentioned in the previous section. Specifically, object-oriented frameworks can be seen as components in an SPL [8]. A clear taxonomy of these concepts is not very well established in the literature but as the term *software product line* suggests, SPLs form families of products, i.e. the set of possible product instances can be enumerated using a certain configuration model[2]. An instantiated framework, on the other hand, does not generally constitute a complete software product.

[2]If the configuration model allows for component creation, the number of possible products is of course infinite.

Basic techniques

The functions in the framework that are meant to be extended or parameterized, are called *extension points*, or *hot spots* [45]. In white-box (calling) frameworks, the abstract operations of the framework can be seen as the extension points.

The use of a framework should be instructed. These instructions are sometimes referred to as *cookbooks*, consisting of individual *recipes* that describe in an informal way how a certain feature or function of the framework should be used and what its restrictions are. [45]

The characteristic feature of calling frameworks is the *Hollywood Principle*, that is "Don't call us, we'll call you". This captures the essence of white-box frameworks in that ideally, the application developer will not need to do anything but provide implementations for the predefined abstract operations, and the framework will do the rest. [30]

The Hollywood Principle most often takes shape as *template operations*, i.e. abstract algorithms defined in terms of one or more abstract operations.

Frameworks are sometimes stacked in the order of decreasing abstractness, so that a base framework provides implementations just for the most generic functions. The subsequent framework layers then implement a certain abstraction layer, providing extension points to the next layers. [24]

Design patterns

An important notion related to object-oriented frameworks is that of *design patterns*. The idea and the resulting *pattern language* were originally developed by Christopher Alexander [1] as a means to describe how to solve a particular kind of architectural design problems.

A design pattern describes a problem that emerges over and over again in a specific domain, and a generic solution to that problem. The essential elements of a design pattern are its name, the problem description, the solution, and the consequences. The name of the pattern should be concise enough to express the basic idea of the pattern and yet be effectively communicated. The problem description explains the design problem in a context, or lists conditions that must be met for the pattern to be useful. The solution is an abstract description of the elements that make up the design, their relationships, responsibilities, and collaborations. Finally, the consequences are the results and trade-offs of applying the pattern. They are useful in evaluating design alternatives. [20]

Generally, patterns are thought to represent ideas at any level of abstraction. However, Gamma et al. [20] define design patterns as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context".[3]

[3]As an aside, there are new object-oriented languages that have some of the most popular

Design patterns' significance to frameworks is that frameworks can be, and often are, designed and documented using design patterns. They also provide useful design vocabulary to facilitate communication about the framework design.

2.3.2 Benefits

Ideally, frameworks provide the infrastructure, or general flow of control, of the application or a component. It is the application developer's job to fill out the missing parts, by specializing the extension points and implementing the abstract parts of template operations, or by creating different combinations of the available objects. This is a significant advantage over traditional program libraries, where the application developer must take care of all the infrastructural details.

The amount of code that the developer must write, test, and debug to develop an application is also generally smaller than in traditional program libraries, because of the infrastructure provided by the framework.

Applications created with a particular framework share the same basic structure. Thus it is easier to maintain the applications, because the same knowledge is valid for many applications. In the life cycle of a framework, the design of the framework also matures and leads to better applications.

When a framework is used, not only the algorithms and data structures provided by the framework classes are reused, but the design and the integrated documentation for the generic parts as well. This enhances productivity of the development effort. Furthermore, as in all successful software reuse, the consistency is improved at every level of the system, from database connectivity to the user interface.

Developing frameworks requires expertise of the domain the framework is targeted to. However, application developers using the framework inherit the domain expertise in the form of the reused design [52]. They only need to concentrate on the details outside the framework's scope. More generally, frameworks make it easier to coordinate the development effort and to assign roles and responsibilities in the development organization.

2.3.3 Problems

Object model limitations

Some authors have proposed extensions to the current object models for alleviating various problems of the framework approach. For example, Batory et al. [6] recognize that object-oriented frameworks are weak with respect to optional features. The

design patterns implemented in the language level. For example, the Ruby language [48] includes implementations for the *Observer*, *Singleton*, *Visitor*, and *Delegate* design patterns. This, in my opinion, is the right direction in rising the abstraction level in software development tools, not different code generation strategies that just widen the gap between the design and the actual (generated) implementation.

rationale is that coding the variations in the framework classes leads to framework proliferation, whereas coding the variations in the application-specific classes leads to code replication and maintenance problems. The suggest that the fixed boundary between the framework and the framework instantiation be relaxed using a technique called *Mixin-layers*.

It can be seen that the authors clearly view the framework as the essence of what we call a product line, suggesting that individual, complete products are built just by extending the framework with the concrete implementation classes and possibly some reusable intermediary mixin-layers that cover some common functionality that was for some reason left out from the framework.

However, when a framework is used within a component-based product line architecture, the configuration model of the product line may provide configuration facilities beyond the scope of the framework *per se*. For example, if the framework and its extensions are distributed over many of the product line components, the choice of the components may provide the concept of optional features that object-oriented frameworks lack. In the next subsection (2.3.4), the various framework component models are discussed in more detail, and later in section 3.2.1, a simple framework component model will be introduced that solves the mentioned problem using the product-line features external to the framework.

Learning curve

Because developing an application framework and developing applications based on the framework are separate disciplines and thus amenable to be performed by different people, the documentation of the use of the framework is of great importance. It is noted that the main obstacle to using frameworks is the learning curve [35]. Unfortunately, the inherent abstractness of frameworks makes it hard to document them. And even when a framework is well documented, which is rarely the case, it is a hard and tedious task to learn to understand the specialization interface.

It has also been noted that "a framework is most useful to someone who understands it in detail" [24]. However, it depends on the framework design and documentation how much of the framework must actually be thoroughly understood by the application developers.

To smoothen the learning curve, Hakala et al. [23] have introduced the concepts of *design contracts* and *specialization templates* as the basic design artifacts and means of documentation for frameworks. Based on these concepts, they have presented FRED, an integrated development and documentation environment for frameworks.

Complexity

The mere complexity of most frameworks make application development based on them laborious. Especially in white-box frameworks, the number of classes that the

application developer must implement is often overwhelming, even if the classes are simple [30].

The inheritance hierarchies and class associations in frameworks are often deep and complex, and as noted by Demeyer et al. [16], they only describe the static relationships of the framework classes, not how the objects actually interact. It is thus hard to understand the inner workings of the framework from the source-code representation.

Architectural Mismatch

Sometimes it is desirable to instantiate and integrate several frameworks and make them work together. Garlan et al. [21] discovered that this situation introduces many types of potential problems. This section is based on their experience report.

In particular, four categories of *architectural mismatch* were identified:

- **Assumptions about the Nature of Components:** Three subcategories fall within this category: (1) *Infrastructure* — A framework may make implicit assumptions about the environment or infrastructure in which it is used. For example, the presence of a certain library may be assumed, although all components would not need it, which may result in awkward and ineffective configurations. (2) *Control model* — A framework may assume that a certain part of the software, be it within or outside the framework itself, holds the thread of control. Or, a certain kind of event loop is provided (or expected) but no means of altering the standard control model in order to be able to inter-operate. (3) *Data Model* — A certain classification of data or a certain set of association types are assumed to hold universally, and no extendability or configurability is available for representing or manipulating data.

- **Assumptions about the Nature of the Connectors:** Within this category there are two areas: (1) *Protocols* — The way of communicating with the framework is often assumed to be uniform. For example, there may not be support for both asynchronous event-based communication and synchronous request-reply style of communication. Having to solve this problem with what one has got may make the communication code complex and error-prone. (2) *Data Model* — The way data is represented in the framework may differ from the representation the framework user needs to utilize. This often requires heavy conversion routines to be used.

- **Assumptions about Global Architectural Structure:** A framework may make assumptions about the global architectural structure in ways that restrict the possibilities of other components.

- **Assumptions about the Construction Process:** It is common that a framework requires some kind of integration code written using the notations

of the framework. Using this code, the rest of the actual application code can use the services provided by the framework. Furthermore, frameworks often employ some kind of a preprocessor or other code generation strategies to create parts of the integration code or other metadata. This affects the development process by dictating the build order, and having two or more frameworks doing the same leads to inconvenience and in the worst case, it forces the developer to manipulate generated code manually or by scripting.

What is not to be inferred from the list above is that a framework should not make this kind of assumptions. On the contrary, in some cases these assumptions are essentially the basis of the architecture. But it is crucial to make the assumptions explicit by documenting them properly. The problem is that although we know how to specify for example the assumptions made by a single interface routine or function, we do not have corresponding conventions or formalisms to express the kinds of assumptions made by frameworks.

Another lesson would be to construct the frameworks themselves using subcomponents that can be changed if an architectural mismatch occurs. This is essentially what object-oriented programming is all about: assigning clear responsibilities to modules and preserving high cohesion and low coupling among them. This is not easy to accomplish and sometimes not even desirable because a trade-off between modularity and efficiency is usually involved.

Lastly, architectural design guidance should be sought. It is not easy to develop understanding about what kind of modules or components work well together. Design patterns (see section 2.3.1) are one way of expressing this knowledge on the scale of object collaboration, but also *Architectural Patterns* have been proposed (see e.g. [46, 47]). Architectural patterns generally impose rule on the architecture that specifies how the system will deal with one aspect of its functionality, e.g. concurrency or persistence [8].

Trade-offs

As identified by Demeyer et al., object-oriented frameworks suffer from an inherent conflict between *reusability* and *tailorability* [16]. In other words, the more generic, and thus reusable, the framework is, the more work needs to be done to create different applications using it. While this trade-off cannot be totally eliminated, its consequences can and should be minimized by proper design.

Another trade-off often present in frameworks is the one between flexibility and efficiency. Or more generally, between abstractness and efficiency, a trade-off observable in many software systems today, for example in graphical user interfaces used in operating systems. By careful framework design the efficiency can be preserved to some extent, but the more abstraction layers are involved, the harder it is to maintain both efficiency and a flexible, modular and safe design.

Note that these trade-offs are general in nature and reflect the ones discussed in the

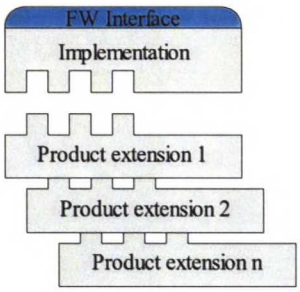

Figure 2.1: Product-specific extension model

context of software product lines (see section 2.2.2).

2.3.4 Framework Component Models

Software product line architectures often involve components, as explained in section 2.2. However, object-oriented frameworks are often used as components or building blocks in software product lines. This section discusses four different framework components models that describe the framework extension mechanisms in the context of a software product line, as identified by Jan Bosch [8]. The illustrations of the extension models are also adapted from his work.

Product-specific extension model

Traditionally, frameworks are extended for each instantiation, or product, that is generated. In the context of a software product line, this results in an instantiation of the framework for each of the products that include the particular component. This organization is illustrated in figure 2.1.

The strength of this model is simplicity; a relatively simple organization of software development will be able to manage it. The main disadvantage is the lack of reuse of the commonalities between the product-specific instantiations (recall the discussion in section 2.3.3, regarding the object model limitations of frameworks). In addition, changes to the framework affect all instantiations, which makes the model highly inflexible.

Note that this model defines a rigid and clear boundary between the framework and the extension. There are no intermediary layers of abstraction in the model. The same applies to the Standard-specific extension model, discussed below.

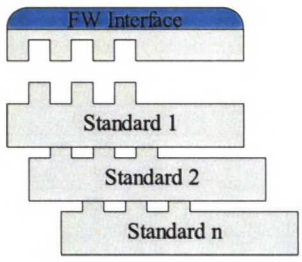

Figure 2.2: Standard-specific extension model

Standard-specific extension model

In the standard-specific model, each standard, e.g. communication protocol or file-system standard, is implemented as an extension to the framework. Each product then incorporates one or more of the framework implementations, depending from the component configuration of the product (see figure 2.2). In this model, the framework component merely defines the interface, and the instantiations are responsible for the whole implementation.

The primary advantage of this model is the uniform interface to the various standard-specific implementations. Like the product-specific extension model described above, this model is conceptually simple and easy to organize.

The standard-specific extension model suffers from three main disadvantages. First, because the common part of the component only defines the interface, the reuse potential of object-oriented frameworks is not exploited. Second, the component model does not allow for product-specific extensions, e.g. wrapping the functionality of the standard in some product-specific fashion. Third, the model is highly unsuited for changes to the component interface, enforced by client components.

It must be noted, however, that the restriction regarding the interface-only nature of the framework in this model is somewhat artificial. For some reason, Bosch omits the possibility that the common parts of the standard implementations were included in the framework. While this restriction may be adequate for some situations, e.g. if the implementations of the standard comes from a third party, there should be no reason to generalize.

Fine-grained extension model

The models discussed above are based on the idea of instantiating the framework with a single extension. The fine-grained extension model takes the opposite approach, i.e. it aims at providing small extensions that only cover one or a few variation points in the framework and that themselves may be extendable. This idea is illustrated in figure 2.3. The base framework component consists of an interface and the implementation common to all instantiations. For each variation point, there is a set

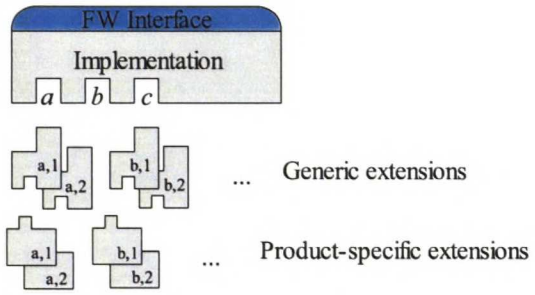

Figure 2.3: Fine-grained extension model

of generic extensions and generic extensions can be configured with product-specific extensions.

The fine-grained extension model is flexible and allows for high reusability and independence of the extensions. The user of the framework is free in composing arbitrary sets of extensions when instantiating the framework.

The prime disadvantage of the model is complexity. Depending on the number of variation points, the number of extension and the number and complexity of the relations between them, the model can be complex to use. In addition, the relations between extensions for different extension points are often implicit, which may make the model too cumbersome to use properly.

Note that this model is a generalization of the notion of layered frameworks stacked in the order of decreasing abstractness (see section 2.3.1).

Generator-based model

The last model for using object-oriented frameworks as components in a product-line architecture differs significantly from the other models presented. The generator-based model is a generalization for schemes that require the use of a tool to configure or instantiate the framework. The tool support may vary from the use of domain-specific configuration languages to graphical configuration tool. Common for these schemes is that they are based on configuration input, the base framework, and a generation process that constructs the actual instantiation. This is illustrated in figure 2.4.

The generator-based model has the advantages of the fine-grained model, namely high reusability and flexibility and logical integrity of extensions. They are achieved because the same fine-grained extensions can be used internally in the generation process. In addition, any framework functionality not used in a particular instantiation can be pruned from the product.

The generator can also be equipped with intelligence about the relations between the extensions for different extension points, and thus proactively simplify the otherwise

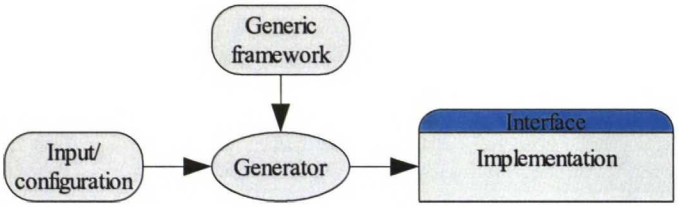

Figure 2.4: Generator-based model

complex use of a framework with fine-grained extensions (see above).

This model has major disadvantages as well. The framework is more expensive to evolve, because changes in the framework must be appropriately reflected in the configuration tool, or more generally, the generator. Similarly, when incorporating product-specific changes, the extension needs to adhere to a protocol defined by the generator, which introduces an overhead to all extension activities beyond simple configuration.

In any of the extension models presented above, Bosch does not address how the extensions fit into the software product line's component architecture. A general framework extension model, that is incorporated in the product line architecture and takes this aspect into account as well, will be presented in section 3.3.

2.4 Object-relational mapping

As explained in section 2.3, object-oriented frameworks often aim to cover some technical domain. In this section, a popular example of such domains, *object-relational mapping*, is examined.

Object-relational mapping, or *OR mapping*, relates to the previous parts of this thesis in two ways: First, it is a common example of a software domain suited to implementation using object-oriented frameworks, discussed in the previous section. Second, as a software domain it is reasonably well understood, and provides an excellent example of a standard architecture for public component market. Recall from section 2.1.3 that software architectures are defined for three different purposes: for describing a single software system, a software product line or a standard architecture for public component market. This relation will be reviewed in the introduction to persistence layers in section 2.4.3.

2.4.1 Background

Although more and more systems are built using object-oriented technology, object-oriented database management systems (ODBMS), object-relational database management systems (ORDBMS) or native XML database systems have not reached the

popularity of relational database management systems. Relational database management systems are still so commonly used for three reasons. First, they often exist in legacy systems that must be used by new systems. Second, RDBMS technology has been available for decades and is well understood. Third, the relational model is simple and has a sound mathematical foundation. [10]

In this work too, a RDBMS is assumed as the underlying data store. But building an object-oriented framework on top of a relational database management system introduces a problem: how should memory-resident programming-language domain objects be represented in an underlying relational database?

As is familiar from basic database systems literature, the relational model deals with *relations*, *tuples*, and *attributes*. In practice, because of the wide adoption of relational database management systems, they are often called *tables*, *rows*, and *columns*, respectively [18]. An obvious conceptual mapping from these concepts to a class-based object-oriented world is to map them to *classes*, *objects*, and *member fields*. However, the relational data model falls short with the standard object-oriented concepts such as *inheritance*, *polymorphism*, and *member operations*. The models also have mismatching interpretations for *identity*: identity is an inherent property of an object but for tuples, it is an externally defined constraint specified by a *key attribute*. Consequently, object associations using references to other objects is not possible for tuples, but *foreign key attributes* must be used. Furthermore, tuples may have atomic attribute values only, whereas objects may have arbitrarily complex structured objects as member fields.

2.4.2 Mapping Concepts

So how can the two models be integrated? Clearly, neither can just adapt the properties of the other model as such.

Identity

A common first choice is to attack the identity mismatch using *object identifiers*, a technique recognized as a Pattern by Brown and Whitenack [10]. Object identifier, or OID, is a fabricated primary key column in every table representing a domain class. This OID is also present as a member field in the corresponding objects, and is thought as the 'domain identity' of the object that will be used to identify it in the query and update operations (the actual object identity as provided by the programming language is of course different).

Using object identifiers as described above is only possible when the underlying database schema can be changed, if it is not being designed from the start.

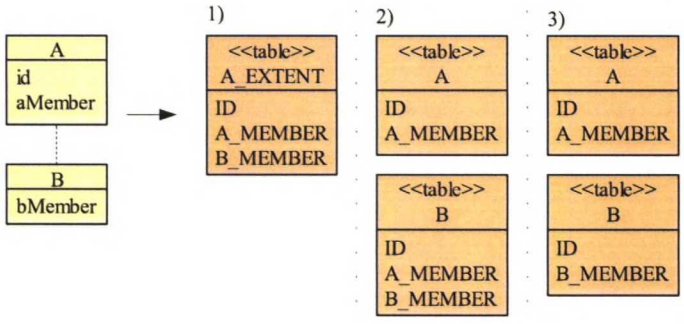

Figure 2.5: Mapping class inheritance to the relational model

Operations

Emulating member operations in a relational database is possible for database procedure languages that support procedure variables, but procedures are not part of the RDBMS standard query language SQL. Polymorphism is naturally out of the question for RDBMSs. However, since an object-oriented database system is not used, the database system can just be viewed as the data store and all of the operations can be written in the actual program code, where object-oriented features are available.

Inheritance

For mapping inheritance, three approaches are available. Consider classes A and its subclass B. A has members id and $aMember$. B has a member $bMember$. Figure 2.5, the classes and their mapping options (explained below) are illustrated.

The first approach is to map both of these classes onto one table, `A_EXTENT`, having columns `ID`, `A_MEMBER` and `B_MEMBER`. Extent means here the set of all objects that coerce to A, i.e. instances of A and B. The drawback of this approach besides high coupling is that the runtime type of the objects cannot be inferred from the tuple data alone, but some extra type information must be used. In particular, a *null* value in the `B_MEMBER` does not imply a runtime type of A, because null values may be allowed for the bMember field for B instances.

Alternatively, each class could be mapped to a distinct table and have all inherited attributes defined redundantly. This would lead to two table definitions, `A` and `B`, where `A` has the attributes `ID` and `A_MEMBER`, and `B` has the attributes `ID`, `A_MEMBER`, and `B_MEMBER`. With this approach too, it is hard to infer the object design from the schema [10].

The last approach is to map each class to a distinct table, but not replicate the inherited attributes except from the ID link. This yields two table definitions `A` and `B` where `A` has the attributes `ID` and `A_MEMBER`, and `B` has the attributes `ID` and

B_MEMBER. In this case, database joins (or multiple queries) are necessary to fully materialize objects of class B. This approach is suggested by Brown and Whitenack in situations where the ease of schema modification is more important, and the former when the speed of the queries is more important [10].

Associations

To map object associations, five kinds of relationships need to be considered, namely 1-to-1, 1-to-N, N-to-M, N-ary, and qualified associations. Continuing with Brown & Whitenack's presentation, they should be mapped as follows.

1-to-1 associations should be represented as an association table, when the relationship itself has a logical meaning or any attributes of its own. When this is not the case, e.g. for a simple 1-to-1 containment, the association should be merged into one of the tables as a foreign key reference. However, it must be noted that either approach does not restrict the association in the database schema to 1-to-1 unless further constraints are used.

For 1-to-N associations, the 'N-peer' of the association should have a foreign key reference to the 'container' table. An N-to-M association should map to an association table with two foreign key references to the both domain tables. N-ary associations, meaning a single relationship that associates N entities together, and all qualified associations should map to association tables having foreign key references to all associated tables.

Collections

As the last mapping-related issue, let us consider the representation of "Collection" subclasses, a common case in Object-oriented languages. Because the first normal form rule of Relational Databases prevents a relation from containing a multivalued attribute [18], the Collection types should be represented in special relationship tables that have a foreign key reference to the containing object, the primary key of the collection item and possibly additional attributes reflecting other attributes or properties of the collection, such as ordering. The primary key of that relation table would then be comprised of the container foreign key and the collection item's identifying attribute.

2.4.3 Implementation

When the model-theoretical approach to OR mapping has been chosen, the implementation issues must be addressed. The OR mapping implementation is usually called a *persistence layer*[4] or *OR mapping tool*. Persistence layers come in many

[4]We stick to the term "layer" here, because in this context it is more widely used and there is no danger of confusion with respect to neither the tiers nor layers as defined in section 2.1.2.

flavors: there are many commercial OR mapping tools (see [4] for a thorough comparison) and open source persistence frameworks such as Object/Relational Bridge (OJB) and Torque (see [41, 54]; a useful list can be found at the DMOZ open directory project [40]). In addition, there are many home-grown custom implementations, such as the one being introduced in this thesis.

There are also interface specifications that define the persistence operations' syntax and semantics, such as the Java Data Objects specification [27] and the more general ODMG3.0 specification [38]. For example, the OJB framework mentioned above implements both of these specifications. These specifications make prime examples of standard architectures defined for enabling a public component market in a well-understood domain (see the discussion about the uses of software architecture in section 2.1.3).

Techniques

The OR mapping implementation, or persistence layer, can be implemented basically in three different ways, as suggested by Scott Ambler [2].

Firstly, the needed SQL statements can be hard-coded in the domain classes. This approach is called *direct mapping* by Craig Larman [32]. It has the advantage that it allows for rapid prototyping or implementation of small applications. The disadvantage is that it leads to replicated and often inconsistent code, and directly couples the domain classes to the underlying database schema, resulting in maintenance problems. However, if the database access code is generated and injected into the class by a post-processing compiler so that the developer never has to see or maintain it, the approach may be workable.

The second kind of persistence layers employ a slightly better approach in which the SQL statements for the domain classes are encapsulated in one or more 'data classes'. In Larman's terminology, this is called *indirect mapping*. It eases maintenance somewhat but the basic problem remains: the classes must be recompiled after every change to the underlying database schema.

Thirdly, the persistence layer may be based on metadata, that is, an external persistence mapping schema. Exemplifying the third kind, Ambler proposes a design for a robust, transparent persistence layer that maps objects to relational databases in such a manner that minor changes to the relational schema do not affect the code. The persistence mechanisms are highly abstracted and the application programmer does not even have to know that the objects are being stored in a relational database. The obvious disadvantage of this approach is the decreased performance due to added abstraction layers. In addition, Ambler's design does not address inheritance in domain classes.

Chapter 3

Alfred

This chapter presents Alfred, a software product line developed in-house by Oikeat Oliot Oy. The presentation continues on the road marked by the previous chapter: from software product lines via object-oriented frameworks to the framework implementation of a persistence layer.

3.1 Background and Overview

At the time of writing, Alfred is three years old and has undergone plenty of transformations and refactoring measures. In this section, the history and motivation of the project is discussed, followed by an overview of the architecture.

This thesis is written for Oikeat Oliot Oy, a Helsinki-based software company with 10 employees, specialized in tailoring technically challenging large scale applications. Oikeat Oliot is purely a software consultant organization; it does not produce commercial off-the-shelf (COTS) software products.

As the main customers of Oikeat Oliot are domestic governmental organizations with similar computational needs, the idea of software reuse had been growing for some time in the company, along with the domain expertise. But it was yet to be properly realized.

Alfred started in the summer of 2001 as a prototype designed and developed for participating to a bid contest set out by a Finnish governmental organization in order to produce a new diary system. The project team consisted of two persons and the head architect, who also informally acted as the project manager.

The original idea behind the work was to produce two stacked frameworks, with a specific application layer on top. The first framework, then called the System Framework, would lay the common foundation for web-based applications, providing e.g. an authentication scheme, logging services, relational database access etc. The second framework, originally called the Application Framework, would then provide common features of a diary system, by providing the basic domain classes with a

generic user interface paradigm providing default screens for the actual application. The design of the System Framework was based on the following assumptions:

1. Simple manipulation of hierarchical domain data residing in a standard SQL database is an appropriate basis for a class of multi-user data management applications.
2. The most commonly used common web browsers have the potential to be adequate clients for such an application, despite the current status of web interoperability and scripting standards and the expense of using a stateless communications protocol, HTTP. This assumption applies primarily to intranet arrangements.
3. The database schema to be used in the application is created concurrently with the framework specialization, or it can be altered so that it becomes subject to certain external constraints.
4. Extending or changing the domain data model and related functionality will be a natural part of the applications' life cycle.
5. Applying modern distributed systems technology and related component models adds too much complexity for the purpose.

These assumptions led us to a design that makes it simple for a developer to define compound domain entities, and provides the users with an efficient and elegant default implementation for manipulating domain data, from the user interface down to the physical database.

But soon it was noticed that this conceptual model of the software architecture was way too simplistic for its intended power and caused severe configuration management problems in practice. The stacked frameworks model then evolved, through various stages, to a full-blown product-line architecture with a configuration model based on component selection, extension and creation.

The technologies underlying Alfred were chosen for pragmatic reasons. Even without deeper insight into current computing environments in organizations, it is safe to say that currently it is common for organizations to have a TCP/IP-based intranet and PC workstations on the employees' desktops. The HTTP-based world wide web is in widespread use. Furthermore, if there are any organizational information systems available, there usually exists some kind of a server computer with an RDBMS installed, either in the organizational unit or somewhere within network access. In most cases, there is a JDBC driver readily available for these as well. These observation are crucial because from a customer's point of view, a solution that is available without further investments on third party software licenses or hardware is very attractive. This was one of the basic design principles of the proposed architecture.

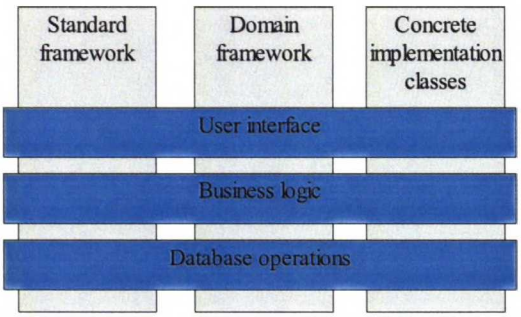

Figure 3.1: Tiers and layers

As the implementation platform we chose the Java Servlet technology, using the Tomcat servlet engine. We used IBM DB2 as our development database engine but any JDBC-compliant database engine can be used as the underlying RDBMS.

The key idea in the design was to define a generic and persistent object model for domain objects, and implement a simple access system for it as an RDBMS-backed web application. This domain object model would not cover the implementation class hierarchies, and use a relational database to persist them, but actually extend the standard extension/inheritance mechanism to the database schema, which would not just allow subclassing of the domain classes (business tier) but also the corresponding database tables (database tier), and of course, the way they are presented to the user (UI tier). In each of these tiers, the framework provides various abstraction layers on top of which the application developer adds the top-level, concrete implementation layer (where the default implementation is inadequate or non-existent). Thus, we have two perpendicular logical stacks, one of layers and one of tiers. This idea is illustrated in figure 3.1. The employed persistence solution is described in more detail in section 3.4, reflecting to Ambler's persistence layer classification (see section 2.4.3).

This "persistence layer with a common user interface" is implemented as a web application product-line architecture consisting of several object-oriented framework components with specialization interfaces and extension points. The framework components include a persistence framework that is a sophisticated application of the *Type Object* design pattern, and a user interface framework that acts as a client component for the persistence framework. These frameworks and other components are described in more detail in section 3.3.

The underlying product line architecture is used to configure, test, build and deploy the application and facilitate the application development process with scripts, templates and examples. The product line features are presented next.

3.2 Alfred Product Line

The core of the Alfred product line deals with the component model definition. Its other responsibilities are the common development infrastructure including source code organization and version control, source code safety checking capabilities using JLint [29], unit testing infrastructure using JUnit [31], product deployment and packaging functions for supported operation platforms and facilities for product development through component creation.

In the next subsections, the product-line architecture is described starting from the description of the Alfred component model. Following that, the development of the product line and the deployment of Alfred-based applications are discussed.

3.2.1 Component Model

In section 2.3.4 we discussed various framework component extension models. In this section, a new extension model is presented as a feature of the Alfred product line architecture, based on Bosch's theory.

The component model of the product line consists of a standard for source code organization and scripts that perform various tasks described in other subsections in this section. The specified technical interface is not related to the actual required and provided interfaces between the product line components.

A component, as defined by the product-line architecture, is a named directory residing in a specific subdirectory in the product line's source tree[1]. The names of the components obey the same naming standard as the Java package name notation, except that the product line's standard components and frameworks are prefixed with the package prefix `alfred`[2]. However, to avoid lengthy component names, the inverted domain name is omitted for Alfred's standard components.

The component model does not distinguish between a framework component and a regular component, just as there is no fundamental difference between a concrete Java class and its subclass. Thus, they generally can always be extended by another component or a subclass, respectively. However, we will refer to the clearly framework-natured components as *framework components* or *abstract components* and the other components just as *components* or *regular components*. The term *extension component* shall mean a component that extends other components or

[1] As mentioned in section 3.1, one of the motivators of the architectural transformation were the configuration management problems related to the "stacked frameworks" model. While these prolems were probably solvable without transforming into a product-line architecture, it is difficult to organize for application development with a mind-set centered around the framework concept, not the product line.

[2] Because of the evolutionary nature of the product line, it is expected that some applications create their own components that are later on generalized into standard frameworks or higher-level domain-specific frameworks; at that time the component name may change, as exemplified by the Maisa diary system framework presented in section 4.1.

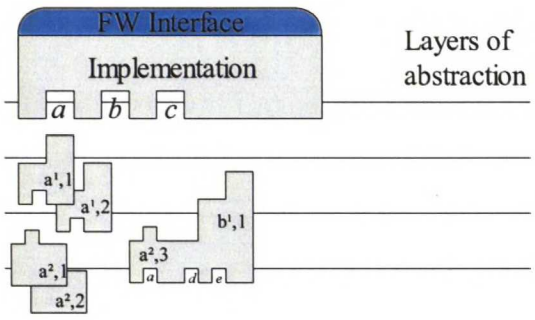

Figure 3.2: Framework extension model developed for the Alfred product line

frameworks[3].

An extension component does not need to extend a single extension point only, as suggested by the fine-grained extension model described in section 2.3.4, but it can extend any extension points in any component, regardless of whether the extension component is a product-specific component or a framework extension component. This property gives rise to the concepts of *horizontal components* and *vertical components*, two special cases of the logical shapes that the components may take in the tiers/layers plane. Figure 3.2 illustrates an example of multilayer, multi-extension-point component, that provides two new extension points (d and e) for more application-specific components.

In contrast to the framework component models presented in section 2.3.4, this model is defined in the product-line architecture, not in the frameworks themselves. The product-line architecture thus provides an outer context, i.e. a management infrastructure for the object-oriented frameworks and other components. This improves the purity of the frameworks, because they need not worry about component integration as they do in a framework-centered model.

Because a product instantiation can freely choose the components included in the build, we now have achieved full support for optional features for frameworks (recall section 2.3.3, "Object model limitations"): there may be an unlimited number of alternative implementations or intermediary domain abstraction levels in the frameworks, freely composable if the framework interfaces are implemented properly.

As an extreme example of this property, in section 3.3 an abstract framework component is presented, where the domain base class, *Object*, has two alternative implementations (extension frameworks) providing different concurrent access policies to suit the application's needs[4].

[3]Bosch [8] refers to this type of components generally as *secondary components*.

[4]Indeed, by component selection it is even possible to define new domain classes in the middle of inheritance hierarchies, not just switch the implementations of certain domain classes.

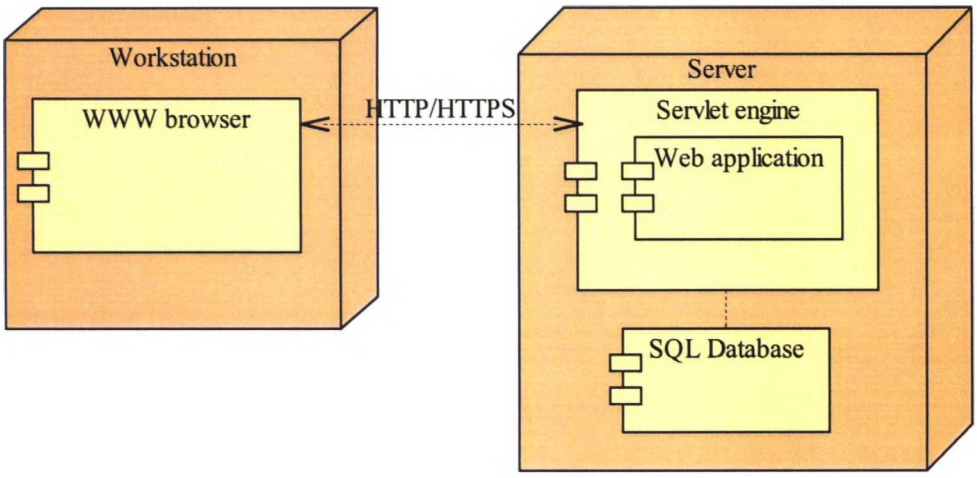

Figure 3.3: A deployment diagram

3.2.2 Deployment

The product line provides a common infrastructure for development-time deployment in a servlet container. Upon deployment, the included components must be selected, and certain platform issues configured. For these tasks, an additional component type definition is needed.

As explained in section 3.2.1, there is no fundamental difference between 'normal' and framework components. However, to be able to test and deploy an application, an *application component* needs to be present. Application components must fulfill the following requirements: a web-application context definition (see the Servlet Specification [50]) must exist in the component's base directory. A file named 'components', listing the included components, must exist in the base directory. The order of the included components must be such that the listed components may only have dependencies to, i.e. required interfaces in, the components already listed. This ensures that there are no cyclic dependencies in the included components.

Technically, the deployment architecture is straightforward. Figure 3.3 illustrates a simple deployment scenario of the Alfred-based products in an UML deployment diagram. Alfred products can be deployed on replicated web application servers and a separate database server, but J2EE-style tier-wise distribution (e.g. web application in one server and business objects in another) is currently not supported.

3.2.3 Alfred Development

The development environment of the product line and its instantiations is based on open-source tools. Any version of any product can be built and packaged basically by checking out the source tree from CVS [15] to a command line environment, and

invoking the build tool GNU Make [34].

The development of the components and frameworks is optimized for the Eclipse IDE [17]. The components, that are subdirectories in product line's standard subdirectory, contain the required metadata for use as an Eclipse Java Project. In the component's base directory, there are predefined subdirectories for the Java source code, unit tests, web application resources, libraries, and SQL batch files.

The product line build system includes Make targets for building, deploying, unit testing, safety checking, and packaging the application at hand. To facilitate component creation, a target also exists for constructing an empty frame for an extension component. In addition, a target is available for automatically setting up a nightly build-test cycle, a popular development-time practice. The resulting batch fetches every night the source code from the repository, builds the product, recreates a test database and populates it with test data, and runs the developer-owned automated tests, sending logs in e-mail in case something goes wrong.

Alfred's development resembled the eXtreme Programming (XP) process [7] in many ways: we had a working system from the beginning on and pair programming was utilised to some extent. The system was maintained in the simplest possible form at all times and aggressive refactoring was not hesitated because a comprehensive test-first-style unit testing infrastructure and was in use, and the development environment and source organization are well suited to change. Design documentation was minimal except for code comments, because the technical design was consensus-based and mainly done in ad-hoc sessions between the developers. The broader vision was also communicated directly, as is only possible with small development teams.

The product line provides a unit testing infrastructure for all its components, both in the command line environment and through the Eclipse IDE. The provided unit testing infrastructure is based on the popular JUnit tool [31], and consists of a regular product line component called `alfred.core.test`. The functionality of the component is to use the product line initialization routines for providing a mock web-application context, which allows unit test style testing of a wider variety of testing levels. For instance, components may define automatic unit tests for a single class, for a package, for a component, or for a whole application.

Between the developers, the work was originally divided to two areas of procession. The design and development from the user interface down was the responsibility of my colleague, and the design and development from the persistence layer up was on my responsibility.

3.3 Components and Frameworks

In this section, a more detailed look is given to the standard product line components that provide the standard functionality and application support to the product line.

First, the component palette and its design principles are presented. Following that, the standard components are introduced individually. Lastly, the interfaces of the components are examined.

3.3.1 Background

The component base's roots, as has already been mentioned, are in the "stacked frameworks" architecture, that actually never worked, except for demonstration purposes. At that time, the core components (see below), comprised the three-tier System Framework and some of the functionality of the other standard components. The Application Framework at that time is no longer part of the standard components, but will be presented as an application example in section 4.1.

The design approach of the core functionality had two directions of procession: from the data model up, and from the user interface down[5]. This means that using web browser scripting capabilities and database schema reflection features, we strived to an architecture that has the advantages of both two-tier and three-tier systems. From two-tier architectures, we have the simplicity of just having to define the user interface and data model, yet being able to add business logic in the middle tier conveniently and in a modular fashion. This is possible because a new technical abstraction level provided by the metamodel of the persistence layer component, that will be discussed in detail in section 3.4; the domain objects are not defined by classes in the Java language, but type objects in the persistence layer, as the *Type Object* design pattern suggests.

The design also provides extension points throughout the architectural tiers, so that specialized behavior can be easily added to the user interface as well as the persistence operations. In particular, the business objects and their operations, being usually at the heart of the system design, are just extension points in the overall framework design that is focused around the data manipulation.

It can also be noted that because of the framework architecture, our approach to object-oriented modeling differed from the conventional approach that emphasizes business object modeling above all. We took advantage of the object-oriented approach for the most areas in the system, including the relational database schema and HTML production, with minimal emphasis on the business object modeling that is the main concern of the actual application developer.

3.3.2 Standard Components

The standard framework components `alfred.ui.web` and `alfred.typeobject`, and the database services component `alfred.db` form the skeleton of a classic three-tier system (see section 2.1.2). These tier components are thus by nature horizontal.

[5]It can be noted that this approach combines the *top-down* and the *bottom-up* conceptual models; the corresponding division of work in the development process was discussed in section 3.2.3.

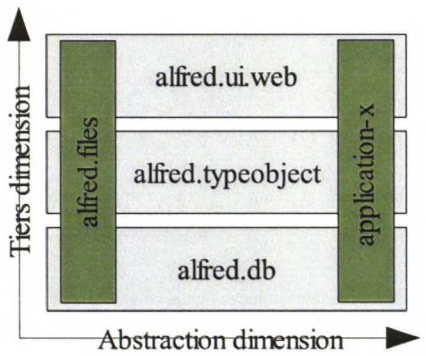

Figure 3.4: Horizontal and vertical components

The other components are more or less vertical in the sense that they operate on different tiers. For example, the component `alfred.files` implements the concept of a 'file' data type. The implementation concerns every tier from the database routines up, through the standard access control and business logic, to the 'upload' and 'download' user interface features. The standard framework components (horizontal), the generic file component and an application-specific component (vertical) are illustrated in the tiers/layers plane in figure 3.4.

The user interface framework supports the application development by providing a powerful yet generic user interface paradigm. It maintains a tree of accessed domain objects, and by using the so-called *formatter* objects and the persistence framework services it allows for a complete CRUD (Create, Retrieve, Update, Delete) life cycle for compound objects as defined by the underlying persistence framework. In other words, the UI framework's core functionality focuses on domain object creation and composition, using generic searching and selection utilities.

The persistence framework `alfred.typeobject`, described in more detail in section 3.4, uses the *Type Object* pattern to define the common properties of the domain objects, most importantly their persistence functions.

The database services component `alfred.db` forms the bottom tier of the three-tier architecture together with the actual database and its programming interface JDBC (Java Database Connectivity). The component's main functions are the the metadata extraction from the database, and acting as a mediator between the persistence layer and the database. The JDBC interface is not wrapped for the persistence layer, but the component manages the database connections and makes sure that the SQL statements are precompiled and cached for maximum performance when used in other components. This is an interesting feature because even though the SQL statements are created dynamically by the persistence framework, the amortized performance compares to the one reached with static SQL queries when the connections are reused.

`Alfred.object.exclusive` is a major secondary component that defines a *pessimistic*, or exclusive access model to the business objects. This means that the first client to request an object (representing a joined tuple of the class-defining tables) will get a read-write access handle, while the rest will only be granted read-only access. After the read-write access is released, the next client will again be able to make modifications to the requested object. This component is itself a framework component, extending the core framework of the persistence layer, `alfred.typeobject`.

The component `alfred.object.shared`, provides concurrent access control in an *optimistic* fashion, which means that read-write access is granted to everyone and timestamp checks are used to ensure that no data is lost in dirty write operations. Of special interest about these components is that they define alternative implementations for the most fundamental of the domain classes, *ObjectClass*, which illustrates the power of the type object pattern combined with the product-line's flexible component model. Of course, nothing prevents the application components or frameworks from providing their own implementations for the root class.

Not included in the core meta model, the component `alfred.codes` implements the concept of multivalued attributes through specialized code tables, where code values are associated with the actual attribute values. In practice, the actual attribute values may be e.g. names of some domain concepts in different languages. The ICodeSet interface is useful in user interface generation, for example when creating a drop-down box for selecting values.

`Alfred.tabular` is another vertical component that implements a simple access paradigm for data that is tabular by nature. Based on this framework component, the application developer may easily construct user interfaces for maintaining various lists or tables, again by extending the appropriate extension points in the component's classes.

The authentication and authorization component `alfred.auth.db` provides an authentication and authorization back-end based on the system database. The two authentication front-end components, `alfred.auth.form` and `alfred.auth.clientcert`, implement different authentication schemes as their names suggest: the former can be used to build a simple login screen with username and password input fields, whereas the latter uses the SSL client certificate to authenticate the user.

The application initialization and some standard services such as loading the language resources and initializing the logging service are performed by the component `alfred.core`. These services are utilized both by the application server at runtime and by the unit testing infrastructure component `alfred.core.test`.

3.3.3 Component Interfaces

Although the importance of the explicitly defined component interfaces was deemed critical, and indeed is a part of the definition being used for components (see section 2.1.1), the explicit definitions of the interfaces are left out of the scope of his

thesis. Ultimately, this work does not aim to be the definitive documentation of the components. Furthermore, at the time of writing, the frameworks and components are in a transient state due to the ongoing development, and because the number of developers of the product line is only two, it has not been necessary to maintain documentation other than in Javadoc [28] format. For this reason, the required and provided interfaces are simply specified in the source code as follows.

The *provided interfaces* of the components are defined in the Java package whose name matches the name of the component. The Java classes and interfaces declared public in this package form the provided interface of that component.

The *required interfaces* of the components are also explicitly defined in the code: any reference to a Java package in another component is guaranteed to appear in the import statements found in the actual program code. To maintain this information anywhere else would be redundant, but a list of the required interfaces can easily be generated from the code.

Note that a typical application component extends more than one framework component in more than one abstraction layers, as allowed by the product line's component model (see section 3.2.1).

3.4 The Persistence Framework

Perhaps the most interesting one of the Alfred's standard components is the persistence framework `alfred.typeobject`. In this section, it is described in more detail, reflecting to the OR mapping background presented in section 2.4. First, a domain data model is defined, following the description of the OR mapping strategies and the actual persistence layer implementation. Finally, the persistence framework's relation to other framework components and to the whole product line is discussed.

3.4.1 Domain Concepts

The development of the standard functionality, despite its generic nature, is guided by the actual targeted application domain, i.e. diary systems. While keeping that in mind, let us describe an idealized and restricted data model for a generic data management domain:

At the basic level, we have *objects* with some data. Other objects may relate to the basic objects either through *aggregation*, establishing a 1-to-1 or 1-to-N association, or by *reference*, establishing a 1-to-1 or N-to-1 association with the parent object. By definition, a N-to-M association will follow when these two types of associations are combined. Similarly, other objects can be associated to these objects and so on.

The *class* of an object defines what kind of relationships the corresponding objects may form. The domain data to be managed consist only of different kinds of objects, structured hierarchically as defined above. The actual data in the objects, the

attributes, can be numeric, textual, or arbitrary binary data.

The set of all objects may be partitioned into disparate *object spaces*. In this scheme, no objects have associations across object space boundaries. Typically, the object spaces are represented in the application by some domain concept, e.g. a single office of an organization.

The data objects are managed by *users*, who belong to *groups* through the notion of *roles*. That is, roles define the N-to-M association between users and groups. By default, every group is associated with one of the object spaces. Every user has a *language* he or she uses within the system.

A *file* is a special kind of an object that has a name and content of variable size and representation.

3.4.2 OR Mapping

The OR mapping strategy plays in a crucial role in the Alfred product line, and was one of the corner stones of the original design. Its design principles included efficiency and ease of modification and extension of the domain data model.

Considering the inheritance mapping options listed in section 2.4.2, it can be seen that the first strategy, using extent tables, is inappropriate because new 'subtables' (inherited tables) cannot be added without modification of the existing extent tables. The third option, table per class, is chosen over the second option, because of the ease of modification and the elegance of not having redundancy in the model. The potential decrease of performance in the database queries was estimated to be less critical and compensated by the efficient persistence mechanism.

Inheritance in the database schema is arranged by defining new tables, whose OID primary key is also a foreign key reference to the parent table. Inheritance hierarchies of arbitrary depth can be introduced in this way. The most fundamental of the database tables is the *object* table, which is the base table for all entities in the domain data model and corresponds to the *Object* class in the Java language.

3.4.3 Persistence Layer

The basic idea of Alfred's persistence layer is that the data objects are only defined at the database back end and their use at higher levels. The JDBC API is used to generate all the persistence functions based on the JDBC metadata, and the persistence layer provides the basis for user interface functions, used by the user interface framework. For defining the objects' associations, however, the foreign key references defined in the database are not enough. Thus they are defined in the framework's extension components along with other properties of the domain classes.

Why did we implement our own persistence layer, when there are plenty of well-tested and widely used persistence frameworks readily available? Firstly, in the

framework design, a dynamic OR mapping solution was a central issue, but just as important was to build a consistent and integrated chain of data all the way from the UI tier downwards, to preserve simplicity and efficience. Secondly, the aim was not to develop a functionally complete persistence layer, but to evolve it simultaneously with the user interface paradigm and its features, while making sure that the efficiency of the database operations is preserved. The same reason explains, why a full-blown persistence broker implementation (see section 2.4) was not even attempted. Lastly, when depending on a third party component, a risk with respect to maintenance and further development of the component is involved. The learning curve of just using a third party component is always significant, but it requires special talent and a lot of effort to maintain or make adaptive changes to an open-source third party component in case the developer community stops developing the component or takes a turn to an undesirable direction. For these reasons, a third party OR mapping solution was not considered.

The design of the persistence layer, or OR mapping implementation, is best described using an implementation concept called a *Bean*[6]. Beans represent the persistence-enabled business objects of the Alfred-based applications and implement the base interface IBean.

The Bean concept was originally implemented an abstract technical services class that defined or generated the database operations using database metadata and the extension points implemented by the domain Beans[7]. However, this introduced a problem. How can the domain class specific properties be inherited into domain subclasses? The Java language does not provide any means to do this. The static methods in Java are not polymorphic. This bottleneck initiated a massive refactorization that ultimately lead to the use of the Type Object design pattern.

The basic idea of the Type Object design pattern is that the domain classes and their instances are not represented as classes and instances of the implementation language, but instead, there is one implementation class for a domain *class* and one for the *instances* of the domain classes. This way, the instances of the domain classes are actually first class Objects in the implementation language, and thus subject to polymorphism and all of the useful features of Object-oriented languages. For instance, static members of the classes defining the domain classes can now be used to specify properties or behaviour common to all domain classes instead of all domain objects. This means that in the business tier the dynamically created domain objects themselves serve primarily as containers of cached database tuples, not domain objects in the traditional object-oriented sense.

[6]This "Bean"is not to be confused with Sun's JavaBean or Enterprise JavaBean specifications [26, 19]. The name was chosen, because the Alfred Beans have a similar logical interpretation as the 'real' JavaBeans in that they have standardized getter and setter interfaces and a certain pattern of construction and lifecycle operations.

[7]As noted by Larman [32], this approach suffers from the weakness of coupling the domain classes with a techical services class – two quite different concerns. At the same time, he emphasizes that "separation of concerns" must not be established at all costs, if the approach leads to an easy and maintainable solution.

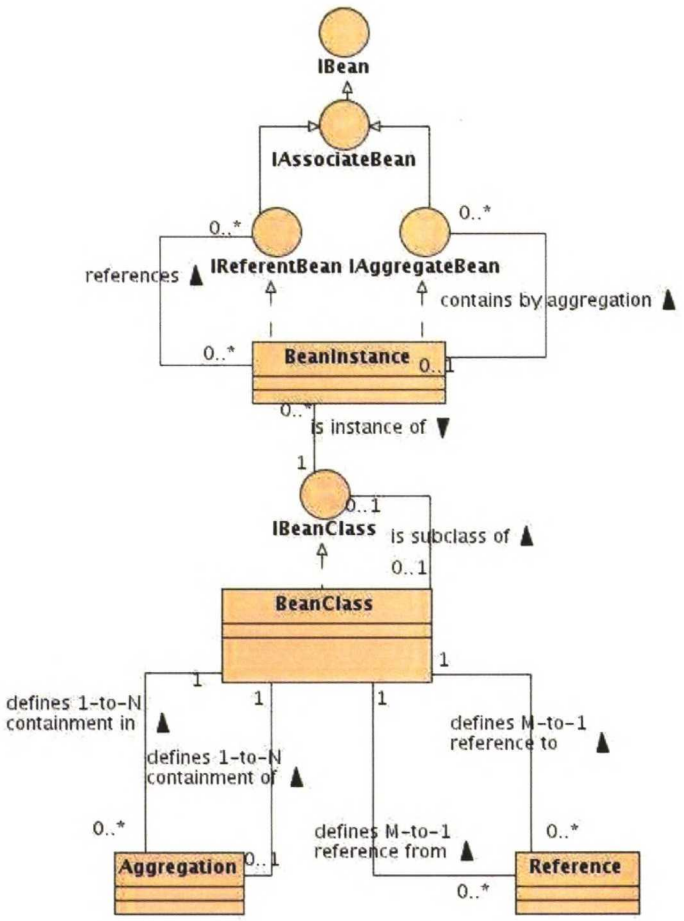

Figure 3.5: Implementation of the Type Object pattern

These basic classes of the persistence solution were named *BeanClass* and *BeanInstance*. The most important classes and interfaces collaborating in the domain object model are illustrated as an UML class diagram in figure 3.5. The chief advantage of using the Type Object approach, in contrast to the former one, is that the new BeanClass definitions for domain objects are very simple, and have no 'magical code' whatsoever (e.g. obligatory static methods). They are defined by either instantiating the default BeanClass or a custom BeanClass subclass, and providing the parent BeanClass as a constructor parameter.

In Ambler's type catalog of persistence layers (see section 2.4.3), this framework falls into the third category: the mapping is based on metadata, no SQL statements are needed for introducing a new domain class. The persistence functions generated using the database metadata are generated and used in the persistence operations by the BeanClass. All abstract and concrete domain BeanClasses implement the IBeanClass interface and inherit the default behaviour from the abstract BeanClass. By default, the instances of the domain classes are instances of the concrete implementation class BeanInstance, that refer to their defining class's methods when their class-specific operations are called. However, the model explicitly allows for custom BeanInstances to be used by any specialized BeanClass, or further subtyping of the domain object instances. Two examples of this are the alternative implementations to the ObjectClass's access control policy (see section 3.3.2) and the ongoing work in introducing a BeanClass with BeanInstances equipped with an XML backend.

3.4.4 Component Organization

As mentioned in section 3.3, the persistence framework is currently covered by three Alfred components: the core of the persistence framework is called `alfred.typeobject`. In addition to that, there are two extension components that define the generic data model defined in section 3.4.1. The extension components, called `alfred.object.exclusive` and `alfred.object.shared`, extend the core persistence framework, providing alternative implementations for the generic base classes of the application domain, most notably the root class, *ObjectClass*.

From the product-line perspective, all of these are just components that comply with the product line's technical component interface. The core persistence framework is not dependent of, and does not need to know about the different extension components that are used with the actual applications.

Chapter 4

Example Applications

The Alfred product line is currently being developed as a basis for a first commercial alfred-based application. Prior to that, a few applications based on the product line have been developed. In this chapter two such applications are presented, namely a diary system framework and an application form delivery framework.

4.1 Maisa

The first Alfred-based application was the actual prototype, called Maisa, that gave rise to the whole Alfred Product Line. The Maisa prototype is what was originally called the Application framework for diary systems, and only later on it was converted to a set of Alfred product line components compliant with the component model described in section 3. In this section Maisa's background and domain are covered, followed by design issues and some future directions.

4.1.1 Background

Governmental organizations often use so-called diary systems to handle their everyday work. Diary systems are information systems that provide the issue and document management features of the diary issue preparation process, from the issue registration all the way through to its delivery to the parties or interest groups. [37]

The Alfred project began when a Finnish governmental oranization set out a bid contest for the production of a new diary system. The system would have to implement many similar features that Oikeat Oliot had been involved with in its other projects. Because of its long experience in diary systems design and maintenance, the company decided to participate.

The product-line approach, even though it was not called that at that time, was appealing not only because of the many reasons discussed in section 1.1, but also because the bid contest included a requirement for a functional demonstration from

the bidders. It would not have been economically feasible to build a working prototype from scratch, because a technology switch was inevitable at that time anyway. The decision was made that the prototype will be built incrementally, based on a simultaneously designed product-line architecture[1].

4.1.2 Domain Concepts

In the domain of diary systems, an exemplary set of basic concepts and their relationships can be roughly defined as follows:

A *case* is a basic object in the diary system domain. The main elements of a case, or issue, are its *parties* and *documents*. Every party of a case is actually a relationship that associates the case with a *person*. Furthermore, *notices* associate specific documents with parties, meaning that the referenced document has been sent to the referenced party.

The handling of the cases is traced using *phases*. The phases of a case are records describing the history of the case, for example, who changed the *state* of the case and when. The handling process of the case types dictates the sequence or network of the possible handling phases at each state of the case.

4.1.3 Design

The implementation of the Maisa clearly calls for a framework approach: there are lots of commonalities between the different case types. Due to this observetion, the original idea of the Application Framework of the diary systems domain was born.

In the Alfred product line architecture, this implies a domain framework component and extension components for each of the case types, deployable independently of each other. Because the concrete issue type example demonstrated was the processing of marriage settlements, the design consists of two Alfred components: `maisa` and `maisa.ms`.

The `maisa` component is the *application component* (see the definition in section 3.2.1). It contains the specification of the needed components to the application, and provides domain-specific classes and interfaces for the actual issue type components. The domain-specific functionality covers both the object model definition and the generic parts of the user interface specifications. This means that a concrete issue type component extends the framework elements in every tier of the system: the generic database tables are 'inherited' by creating a 'subtable' with an ID reference (see section 2.4.2) to the parent table. The domain data classes are extended from the generic framework domain classes, providing the specialized business logic where required. Their associations to other entities (e.g. application – applicant) are specified in the initialization routines. Similarly, the user interface classes (i.e. formatters) typically forward the user interface processing to the more generic

[1] At that time, it was called a *layered framework architecture*.

framework-provided formatters. All of these elements are created as prescribed by the product-line standards.

Note that the same component acts as a domain framework and the actual application component. This is an example of the flexibility of the component model: any extension points can be extended while at the same time providing more extension points to other components and being a concrete application component.

This component organization resembles a *plug-in* architecture in that the base application is functionally enhanced by adding 'plugin' components. The plugin components define their own domain subclasses, business rules and user interface elements e.g. menu items, that are available after redeployment.

The marriage settlement issue type component `maisa.ms` thus covers the marriage settlement specific data types, user interface specifications and business logic.

In practice, having the domain framework act as the application component only makes sense when the evolution and the prototypical nature of the domain framework are taken into account. When a diary system is contracted in the future, a new application component, with dependencies to the diary system framework component, is introduced. Thus, the application component nature of the `maisa` component is essentially a blueprint application for the domain framework.

4.1.4 Further development

The current version of the Maisa framework does not include document management features with the exception of arbitrary files, that can be attached and managed in conjunction with the diary cases. However, sophisticated document management features are being added to the Maisa application. The document production technique allows a WYSIWYG-style draft editing capability by exploiting the "design mode" features and scripting capabilities of the modern web browsers. The documents can contain arbitrary structure and e.g. fine-grained access control policy through the internal XML representation. The company's internal expectations of this technique are high.

Another ongoing development area in the Maisa framework is the process flow. The different possible handling phases, and their availability for the different groups of users, should be specified *declaratively* rather than *procedurally*. This feature could, if a suitable abstraction level is found, even be incorporated to the standard components, so that any Alfred application and not just Alfred-based diary system applications, could use it. This would result to a generic horizontal extension component (named e.g. `alfred.process`) that provides basic functionality and new extension points to process-enabled applications such as the diary system applications based on Maisa.

4.2 Lotta

The second example of the Alfred-based applications is Lotta, an application form delivery system. This section begins with a brief assessment of the current status of the electronic services available for the Finnish public. Following that, the application and its integration with the Alfred product line is described.

4.2.1 Background

Electronic services provided to the public is currently a hot topic in the Finnish governmental information administration. The rapid aging of the Finnish population implies a growing need for personal, social and medical services while no increase in resources is in sight. Thus, there is a need for savings in other areas of government. This implies more efficient use of automated and web-based services. [44]

The current state of practice of electronic services is mainly limited to paper forms available for download in the PDF format. Sometimes these PDF paper forms include a mechanism for filling in the fields and print the readily filled-in form out in the paper, which can then be sent to the appropriate agency for processing. Or the document may be in a format recognized by common text-processing software, and have fields defined for filling in the data before printing. A useful collection of the forms provided for the Finnish public are available in the Lomake.fi service [33].

The paper form is a starting point of the process development. However, it should not be a goal just to bring the paper form to the network in electronic form. More generally, automating old processes does not yield the best possible result. The goal is to renew the service processes. However, this implies a stepwise approach, and the on-line fillable forms are the step that the Finnish information society is currently taking. [43]

In Finland, the Ministry of Finance is responsible for coordinating the development of central government electronic transaction and network services, developing joint services and a joint service infrastructure, issuing guidelines for the development of services and improving access to information [36]. But because each governmental agency is responsible for developing its own electronic services, Oikeat Oliot decided to demonstrate the new technology to a long-time governmental agency client and test the maturity of the Alfred product line.

As the substance of the form delivery demonstration, the application form needed for *property registration* was transformed into an interactive, online application form with extra features such as input validation, arbitrary electronic attachments, and dynamic addition of the aggregate items. The actual paper form used for property registration applications is available from the website of the Finnish Judicial system [42].

4.2.2 Domain Concepts

The application form delivery domain can be coarsely characterized around the basic entity, application. The application is issued by one or more *applicants*. A specific type of form defines the actual structure of the application. For example, a *property registration application* has a *target*, a *yield* (fi. saanto), the *basis* for the application, and one or more *attachments*. As in the Maisa application, there are *phases* that the applicant can follow to see the status of his or her application, after a clerk has processed the application.[2]

It is envisioned that the Lotta framework act as a front-end system and have interfaces to the diary system that is used in the actual processing of that application. For instance, a clerk could search the Lotta system for new applications, and register a new handling phase by transferring the data from the Lotta front-end system to the actual diary system automatically. When the applicant returns to check the status of his or her application, it is visible that the application has been registered and the processing has begun. Thus, the unnecessary work phase introduced by the paper form processing is eliminated. A screenshot image of Lotta is presented in figure 4.1.

4.2.3 Design and Integration

Similarly to the Maisa framework, the various applications and their handling processes have plenty of commonalities. For this reason, the components of the Lotta application follow the same pattern: there is one main framework component and one specialized component for each concrete application type, in this case, property registration. The application development produres are also similar to those of Maisa: the developer needs to provide the data model, database definition and the user interface definitions. We now take a more practical look of what needs to be done on the application's part and what kind of support is provided by the product line.

First, the application skeleton is created in the development tree by commanding 'make new-application'. This creates the required files and directories for the new application component, `lotta`. Following that, the extension component `lotta.pr` providing the property registration functions is created using the command 'make new-component'. The new Eclipse-enabled component directories can then be developed as normal Java projects in the Eclipse IDE.

The next steps are to create the initialization classes for the components. For the component `lotta`, the initializer needs to create the generic bean classes, e.g. the generic ApplicationClass, and its aggregation to the ApplicantClass. The application class contains business logic that applies to all applications irrespective of their actual application type. For example, the ApplicationClass may specify which parts of the applications may be modified by the applicant after the form is first sent to the

[2]See [42] for further explanation of these concepts (in Finnish).

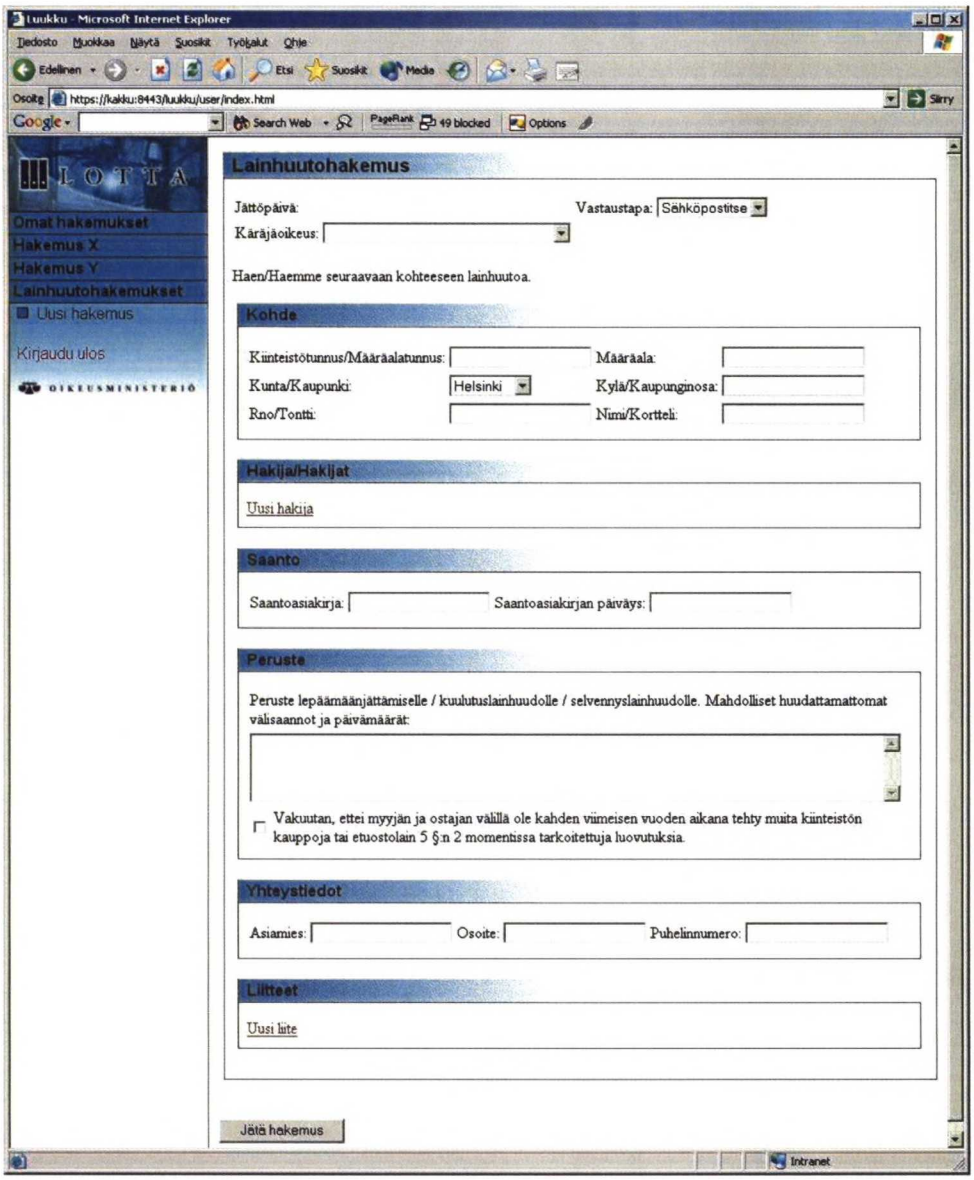

Figure 4.1: A screenshot of Lotta

system.

In the class definitions, the developer does not need to list the data fields of the application objects. Instead, the developer creates standard SQL DDL (Data Definition Language) files that specify the column names and types for each table corresponding to the created BeanClasses in that component. The DDL files, named according to product line rules, serve two purposes: first, they are used by the product line database initialization scripts in (re)creating and populating the database, a frequent development-time operation. Second, they are used in generating the bean class's persistence operations and data containers.[3]

The most involved step is the user interface definition. The framework component `lotta` defines a generic formatter called LottaFormatter that extends the standard BeanFormatter extension points and provides generic convenience methods for user interface generation for the application type specific concrete formatters. The formatting process follows the Template Method design pattern, where the abstract base class does the actual formatting using abstract operations provided by the subclasses.

A typical data management application includes searching facilities and a menu system to help the user navigate in the system. Alfred's standard components provide a basic infrastructure for these as well. Lotta uses these facilities in a combination: the framework component defines a menu item that dynamically displays the applications registered by the current user, i.e. a "My Applications" feature. It is implemented by extending the GenericSearch base class and associating it with a registered menu item. Furthermore, application handlers (see below) have a menu item that allows them to search for new applications to process. This search, unlike the menu search, requires a new searching screen with various search criteria. This, again, works out by using the standard search screen base classes and generic search engine implementations. The appropriate SQL statements are created automatically, joining the data tables that are required with the search criteria present at a given search. According to the framework practice employed everywhere in the standard components, the developer may pick the parts of standard implementations that suit his needs. For example, if the search requires more control than implementable by parameterization, the developer may just choose to implement the ISearchEngine interface instead of extending the GenericSearch class, or just override the SQL generation part of the generic implementation.

Lotta has three user profiles: the anonymous user requiring no authentication, authorized users and application handlers. From Alfred's perspective, these fall into two categories differentiated by the requirement of HTTP authentication and secure transport. These options are indeed preconfigured by Alfred under the roles *user* and *guest*. Because the normal operation of Alfred-based applications goes through a single URL entry point, this is the highest possible level of declarative security (as defined in the Servlet Specification) that can be offered. Other role assign-

[3]This is a prime example of the "Define it only once" principle in the XP process (see section 3.2.3) and Alfred design, which admittedly sometimes leads to other inconveniences.

ments and membership tests are performed in the code. The standard component `alfred.auth.db` defines database tables and a parameterized configuration for performing the authentication and authorization from the system database. When this is not appropriate for an application, it needs to implement the authentication interface, for example using an organization-wide LDAP directory. The first one created will probably be turned to a core asset, under the component `alfred.auth.ldap`.

Chapter 5

Conclusions

In this work, a first, small-scale attempt to develop a software product line was made, with encouraging results. A working product-line architecture was developed, including standard framework components and domain-specific framework components of the core expertise area of the company, specifically diary systems.

The Alfred product line was examined from various perspectives, starting from the product-line architecture, through the standard frameworks all the way to the implementation of a persistence framework component. A new framework extension model was developed as a feature of the Alfred product-line architecture. Two Alfred-based application prototypes were introduced. This section concludes the work by discussing the project in general, then from the organizational and technical viewpoints. Finally, suggestions for further development are given.

5.1 Discussion

In a point where the first Alfred-based customer project is in its implementation phase, it is convenient to discuss the advantages and disadvantages of the product-line approach, both generally and in the context of the actual product line at hand. In the development of a new software system, the expectations and assumptions about the product line's capabilities are weighed, and after some Alfred-based customer projects the project lead will eventually be able to assess the economic impact the product-line approach.

When planning for business and bidding for software system contracts, it is indeed a significant advantage to have a software product line ease effort estimation and lower the total cost, as explained in section 2.2.1. Of course, the production plans become "proven" first after some project realizations, but even in the first project it facilitates the effort estimation, provided that the management has some insight to the capabilities of the product line. In a development organization of this scale, however, it is not likely that the realizations could be scrupulously recorded and compared to the one-off production approach to obtain data about the factual economic

consequences of the product-line approach.

5.2 Organizational Issues

Being mainly an effort of two developers, the Alfred project has not been a conscious organizational shift to the product-line approach, but rather a technical experiment that attempts to package the current best practices of the company. While initiating a software product line should always be a planned effort, it was learned that incremental and iterative design is another possible way to tackle the problems arising from the product-line approach, if the development environment is flexible and the development team small enough to enable efficient communication.

In the recent Alfred development, more attention is being paid to the division of developer roles into Alfred's development and Alfred-based application development. Until now, same people have been doing both. It should be no surprise that without coordination, the developer documentation lags behind in these circumstances. Writing (and testing) that documentation reveals problems or solutions that seem cumbersome when the internals of the system are unknown. A software product line developer writes software for other software developers, and should keep that in mind.

It is not just developer documentation that calls for coordination. The product-line approach clearly requires strategic guidance in order for the development not to get stuck on irrelevant details or simply developing the architecture or the frameworks in such directions that do not actually serve the business goals (that should be made unambiguous) or inadvertantly stretches the scope of the product line.

As one of the developers, it is easy to agree with the product-line-related individual benefits listed in section 2.2.1. Working with the product-line architecture and the object-oriented framework components has been the most challenging and interesting time of my brief professional career. Designing and implementing new required functionality in the core assets, be it a little harder than a one-off implementation, is more rewarding because of the generality. It is essentially solving a problem and multiplying the benefit of the solution with the number of product line instantiations.

5.3 Technical Issues

The application prototypes developed alongside the product line itself tend to utilize the "Alfred way" of doing things. The interesting and more challenging part is implementing a real-world application using Alfred. First in this situation, the flexibility of the product line and the standard frameworks reveals itself in unforeseen ways, as do the shortcomings of the design. For example, unfinished or obsolete feature implementations often require refactoring or further development affecting many parts of the product line. In this situation, a good IDE such as Eclipse assists

The unit testing infrastructure is invaluable in testing the changes, and allows for aggressive refactoring, and system test plans serve other products as well.

- **Portability:** Being based on Java technology and the generic Servlet specification, the product line can be easily extended to support deployment to different platforms, Servlet engines and J2EE application servers when the need arises (see the next section).

5.5 Future Directions

Some of the development ideas related to the existing Maisa framework have already been discussed in the previous chapter. But a few ideas have also emerged for further development of the whole product-line architecture and the standard framework components.

Regardless of the limitations of the original design assumptions (see section 3.1), the applicability of the Alfred product line architecture to build J2EE applications is under investigation. While the current deployment architecture only relies on the Servlet Specification and assembles the specified components into a J2EE Web Archive (WAR), there should be no problem converting the whole build process to utilize other J2EE specifications and perhaps assemble a whole J2EE Enterprise Archive (EAR) as the result. In that case, a likely solution would be to incorporate the deployment process to the supported application servers such as IBM WebSphere, BEA WebLogic or JBoss.

The product build architecture, as explained in section 3.2.3, is based on the GNU Make build tool and the Bourne shell, accompanied by some perl scripts. While these tools are generally available for any platform, the build architecture has not been specifically designed to be cross-platform, as are the builds done with the Java-based Ant build tool [3]. Ant was knowingly declined because of the overhead involved with the use and extension of even the simplest tasks, but this is clearly a change candidate if the transformation to the J2EE platform is initiated.

Another ongoing idea is to generalize the current user interface paradigm further to accommodate an XML-backend for the beans, where the UI-manipulated object structures are input into an XML document instead of the bean hierarchies. The XML content would be stored in the database and upon request, formatted as a web page for further manipulation, or even as a PDF document for a specific presentation format. This model would unify the Beans' association model and the structure of XML documents. In particular, a domain object model would correspond to an XML schema or DTD (Document Type Definition).

Most of the development pressure in the standard components or in the product-line architecture is initiated by some application or domain specific functionality, that seems general enough to be abstracted and incorporated to the generic parts. For example, the XML feature mentioned above has its roots in the Maisa document

management features (see section 4.1.4). A useful property of the component architecture is that there is ultimately no technical categorization of the components[1], just dependencies. This flexibility makes it easy to manage and maintain the component sets and their implemented domains while characterizing and analyzing the design abstractions.

As all successful software, Alfred seemed powerful enough to be applied also to different domains than originally planned. Although the Lotta application form delivery system is conceptually not very far from a diary system, there are many ideas for Alfred's use in quite different purposes, including the domain of the current debut contract.

Based on the experiences so far, it seems that the product-line approach can be quite suitable for small software consultancies, if not necessary for survival against bigger competitors in the tailored systems market. A core experise area or a specific domain that many projects fall into is a good candidate in the search for reuse potential, but a carefully scoped and designed product line may provide competence beyond a single application domain. Furthermore, it should be easy to start a new product line in a different domain once acquainted with the organizational and technical aspects of the product-line approach.

In a way, designing and developing frameworks is the essence of object-oriented programming. During this project it was realized that nothing can teach a designer and developer more about object-oriented development, its benefits and pitfalls, than hands-on development and use of frameworks. But it is the mental shift to the product-line world that reveals a broader view to object-oriented frameworks as software components.

It is now too early to evaluate the success of the Alfred product line as a whole. But as learning vehicles, both the Alfred project and writing about it have been extremely useful. I look forward to experiencing the whole life cycle of the product line and its applications.

[1]Except for the *application component* distinction.

References

[1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel. *A Pattern Language.* Oxford University Press, 1977, ISBN 0-1950-1919-9.

[2] S. Ambler. *The Design of a Robust Persistence Layer For Relational Databases.* <URL:http://www.ambysoft.com/persistenceLayer.pdf> [15.4.2003]

[3] *Apache Ant.* A Java-based build tool. <URL:http//ant.apache.org> [19.3.2004]

[4] Barry & Associates, Inc. *Object-Relational Mapping Product Comparison* <URL: http://www.barryandassociates.com/reports/objectrelational.html> [15.4.2003]

[5] I. Bass, P. Clements, and R. Kazman. *Software Architecture In Practice.* Addison-Wesley, 1998. ISBN 0-3211-5495-9.

[6] D. Batory, R. Cardone and Y. Smaragdakis. *Object-Oriented Frameworks and Product Lines.* Proceedings of the First Software Product Line Conference, 2000, pp. 227–247.

[7] K. Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 1999, ISBN 0-201-61641-6.

[8] J. Bosch. *Design & Use of Software Architectures: Adopting and evolving a product-line approach.* Addison-Wesley, 2000, ISBN 0-201-67494-7.

[9] F. P. Brooks, jr. *No silver bullet: Essence and accidents of software engineering* IEEE Computer, 20(4), 1987, pp. 10–19.

[10] K. Brown and B. Whitenack. *Crossing Chasms: A Pattern Language for Object-RDBMS Integration.* <URL:http://www.ksc.com/article5.htm> [11.4.2003]

[11] E. Casais. *An Experiment in Framework Development.* The Theory and Practice of Object Systems 1(4), 1995, pp. 260–280.

[12] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2002, ISBN 0-201-70332-7.

[13] *Communications of the ACM.* 40(10), 1997. Special issue on Object-oriented Application Frameworks.

[14] *OMG's CORBA Website.* <URL:http://www.corba.org/> [16.3.2004]

[15] *Concurrent Versions System.* <URL:http://www.cvshome.org> [16.3.2004]

[16] S. Demeyer, T. D. Meijler, O. Nierstraasz, and P. Steyaert. *Design guidelines for tailorable frameworks.* Communications of the ACM, 40(10), 1997, pp. 60–64.

[17] *The Eclipse Project.* <URL:http://www.eclipse.org> [23.2.2004]

[18] R. Elmasri and S. Navathe. *Fundamentals of Database Systems.* Addison-Wesley, 3th edition, 2000, ISBN 0-201-54263-3.

[19] *Enterprise JavaBeans Specification* <URL:http://java.sun.com/products/ejb/> [28.4.2003]

[20] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995, ISBN 0-201-63361-2.

[21] D. Garlan, R. Allen and J. Ockerbloom. *Architectural mismatch, or, Why it's hard to build systems out of existing parts.* Proceedings of the 17th International Conference on Software Engineering, 1995, pp. 179–185.

[22] I. Haikala and J. Märijärvi. *Ohjelmistotuotanto.* Suomen ATK-kustannus, 7th edition, 2000, ISBN 951-762-769-6.

[23] M. Hakala, J. Hautamäki, J. Tuomi, A. Viljamaa, J. Viljamaa, K. Koskimies and J. Paakki. *Managing ObjectOriented Frameworks with Specialization Templates.* ECOOP'99 Workshop on Object Technology for Product-line Architectures, European Software Institute, Spain, 1999, pp.87–98.

[24] J. Hautamäki. *A Survey of Frameworks.* University of Tampere, Series of publications A, march 1997. <URL:http://www.cs.uta.fi/reports/pdf/A-1997-3.pdf> [6.2.2003].

[25] I. Jacobson, G. Booch and J. Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999, ISBN 0-201-571692.

[26] *JavaBeans Specification* <URL:http://java.sun.com/products/javabeans/> [28.4.2003]

[27] *Java Data Objects (JDO) Specification.* <URL:http://java.sun.com/products/jdo/> [15.4.2003]

[28] *Javadoc Tool Home Page.* <URL:http://java.sun.com/j2se/javadoc/> [16.3.2004]

[29] *JLint.* A static checking tool. <URL:http://artho.com/jlint> [24.2.2004]

[30] R. Johnson and B. Foote. *Designing Reusable Classes.* Journal of Object-Oriented Programming, 1(2), 1988, pp. 22–25.

[31] *JUnit.* A unit testing framework. <URL:http://www.junit.org> [16.3.2004]

[32] G. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process.* 2nd Edition, Prentice Hall PTR, 2002, ISBN 0-13-092569-1.

[33] *Lomake.fi.* A collection of electronic forms for Finnish citizens. <URL:http://www.lomake.fi> [16.3.2004]

[34] *GNU Make.* A build tool. <URL:http://www.gnu.org/software/make/> [16.3.2004]

[35] M. Mattsson. *Frameworks FAQs.* <URL:http://www.ipd.hk-r.se/michaelm/fwpages/fwfaqs.html> [11.3.2003]

[36] The Ministry of Finance. *Electronic services* <URL:http://www.vm.fi/vm/liston/page.lsp?r=2679&l=en [16.3.2004]

[37] The Finnish National Archive (NARC). *SÄHKE-hankkeen (Kohti sähköistä asiakirjahallintoa) esitutkimus.* A Preliminary Report of the SÄHKE Program, 2001. <URL:http://www.narc.fi/sahke/esitutkimus.pdf [15.3.2004]

[38] Object Data Management Group (ODMG). *ODMG 3.0.* <URL:http://www.odmg.org> [24.5.2003]

[39] Object Management Group (OMG). <URL:http://www.omg.org> [16.3.2004]

[40] *Object Persistence.* A directory entry at dmoz – the open directory project. <URL:http://dmoz.org/Computers/Programming/Languages/Java/Databases_and_Persistence/Object_Persistence> [15.4.2003]

[41] *Object/Relational Bridge.* An Apache DB subproject. <URL:http://db.apache.org/ojb> [15.4.2003]

[42] The Finnish Judicial System. <URL:http://www.oikeus.fi> [17.3.2004]

[43] The Advisory Board for Information Management in Public Administration. *Paperilomakkeista sähköisiin palveluprosesseihin.* The Finnish Ministry of the Interior, 2003. <URL:http://www.intermin.fi/intermin/hankkeet/juhta/home.nsf/pages/B54519A1EB106E03C2256E01004328EA?Opendocument> [16.3.2004]

[44] The Finnish Information Society Advisory Board. *Public Services in the New Millennium: Programme of Action to Promote Online Government.* The Finnish Ministry of Finance. <URL:http://www.vm.fi/tiedostot/pdf/en/40644.pdf> [16.3.2004]

[45] W. Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley. 1994, ISBN 0-201-422948.

[46] T. Richner. *Describing Framework Architectures: more than Design Patterns.* Proceedings of the ECOOP '98 Workshop on Object-Oriented Software Architectures, 1998.

[47] D. Roberts and R. Johnson. *Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks.* Proceedings of PLop'96, Third Annual Conference on the Pattern Languages of Programs, 1996.

[48] *The Ruby Language home page.* <URL:http://www.ruby-lang.org> [31.7.2004]

[49] M. Shaw and D. Garlan. *Software Architecture – Perspectives on an Emerging Discipline.* Prentice Hall, 1996, ISBN 0-13-182957-2.

[50] *The Java Servlet Specification.* <URL:http://java.sun.com/products/servlet> [16.3.2004]

[51] S. Sparks, K. Benner and C. Faris. *Managing object-oriented framework reuse.* IEEE Computer 29(9), 1996, pp. 52–61.

[52] Taligent Inc. *The Power of Frameworks.* Addison-Wesley, 1995, ISBN 0-201-48348-3.

[53] A.S. Tanenbaum. *Computer Networks.* Prentice Hall, 1988, ISBN 0-1306-6102-3.

[54] *Torque.* An Apache DB subproject. <URL:http://db.apache.org/ojb> [15.4.2003]